Record Linkage Toolkit Documentation

Release 0.8.1

Jonathan de Bruin

Dec 22, 2017

First steps

All you need to start linking records.

CHAPTER 1

About

1.1 Introduction

The **Python Record Linkage Toolkit** is a library to link records in or between data sources. The toolkit provides most of the tools needed for record linkage and deduplication. The package contains indexing methods, functions to compare records and classifiers. The package is developed for research and the linking of small or medium sized files.

The project is inspired by the Freely Extensible Biomedical Record Linkage (FEBRL) project, which is a great project. In contrast with FEBRL, the recordlinkage project makes extensive use of data manipulation tools like pandas and numpy. The use of *pandas*, a flexible and powerful data analysis and manipulation library for Python, makes the record linkage process much easier and faster. The extensive *pandas* library can be used to integrate your record linkage directly into existing data manipulation projects.

One of the aims of this project is to make an extensible record linkage framework. It is easy to include your own indexing algorithms, comparison/similarity measures and classifiers. The main features of the Python Record Linkage Toolkit are:

- Clean and standardise data with easy to use tools
- Make pairs of records with smart indexing methods such as **blocking** and **sorted neighbourhood indexing**
- Compare records with a large number of comparison and similarity measures for different types of variables such as strings, numbers and dates.
- Several classifications algorithms, both supervised and unsupervised algorithms.
- Common record linkage evaluation tools
- Several built-in datasets.

1.2 What is record linkage?

The term record linkage is used to indicate the procedure of bringing together information from two or more records that are believed to belong to the same entity. Record linkage is used to link data from multiple data sources or to find duplicates in a single data source. In computer science, record linkage is also known as data matching or deduplication (in case of search duplicate records within a single file).

In record linkage, the attributes of the entity (stored in a record) are used to link two or more records. Attributes can be unique entity identifiers (SSN, license plate number), but also attributes like (sur)name, date of birth and car model/colour. The record linkage procedure can be represented as a workflow [Christen, 2012]. The steps are: cleaning, indexing, comparing, classifying and evaluation. If needed, the classified record pairs flow back to improve the previous step. The Python Record Linkage Toolkit follows this workflow.

See also:

Christen, Peter. 2012. Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection. Springer Science & Business Media.

Fellegi, Ivan P and Alan B Sunter. 1969. "A theory for record linkage." Journal of the American Statistical Association 64(328):1183–1210.

Dunn, Halbert L. 1946. "Record linkage." American Journal of Public Health and the Nations Health 36(12):1412–1416.

Herzog, Thomas N, Fritz J Scheuren and William E Winkler. 2007. Data quality and record linkage techniques. Vol. 1 Springer.

1.3 How to link records?

Import the `recordlinkage` module with all important tools for record linkage and import the data manipulation framework **pandas**.

```
import recordlinkage
import pandas
```

Consider that you try to link two datasets with personal information like name, sex and date of birth. Load these datasets into a pandas `DataFrame`.

```
df_a = pandas.DataFrame(YOUR_FIRST_DATASET)
df_b = pandas.DataFrame(YOUR_SECOND_DATASET)
```

Comparing all record can be computationally intensive. Therefore, we make smart set of candidate links with one of the built-in indexing techniques like **blocking**. Only records pairs agreeing on the surname are included.

```
block_class = recordlinkage.BlockIndex('surname')
candidate_links = block_class.index(df_a, df_b)
```

Each `candidate_link` needs to be compared on the comparable attributes. This can be done easily with the Compare class and the available comparison and similarity measures.

```
compare = recordlinkage.Compare()

compare.string('name', 'name', method='jarowinkler', threshold=0.85)
compare.exact('sex', 'gender')
compare.exact('dob', 'date_of_birth')
compare.string('streetname', 'streetname', method='damerau_levenshtein', threshold=0.
↪7)
compare.exact('place', 'placename')
compare.exact('haircolor', 'haircolor', missing_value=9)
```

```
# The comparison vectors
compare_vectors = compare.compute(candidate_links, df_a, df_b)
```

This record linkage package contains several classification algorithms. Plenty of the algorithms need trainings data (supervised learning) while some others are unsupervised. An example of supervised learning:

```
true_linkage = pandas.Series(YOUR_GOLDEN_DATA, index=pandas.MultiIndex(YOUR_MULTI_
↪INDEX))

logrg = recordlinkage.LogisticRegressionClassifier()
logrg.learn(compare_vectors[true_linkage.index], true_linkage)

logrg.predict(compare_vectors)
```

and an example of unsupervised learning (the well known ECM-algorithm):

```
ecm = recordlinkage.BernoulliEMClassifier()
ecm.learn(compare_vectors)
```

CHAPTER 2

Installation guide

2.1 Python version support

The Python Record Linkage Toolkit supports the versions of Python that Pandas supports as well. You can find the supported Python versions in the Pandas documentation.

2.2 Installation

The easiest way of installing the Python Record Linkage Toolkit is using `pip`. It is as easy as typing:

```
pip install --user recordlinkage
```

You can also clone the project on Github. The license of this record linkage package is GPLv3.

2.3 Dependencies

The following packages are required. You probably have most of it already ;)

- numpy
- pandas (>=0.18.0)
- scipy
- sklearn
- jellyfish: Needed for approximate string comparison and string encoding.
- numexpr (optional): Used to speed up numeric comparisons.

CHAPTER 3

Link two datasets

3.1 Introduction

This example shows how two datasets with data about persons can be linked. We will try to link the data based on attributes like first name, surname, sex, date of birth, place and address. The data used in this example is part of Febrl and is fictitious.

First, start with importing the `recordlinkage` module. The submodule `recordlinkage.datasets` contains several datasets that can be used for testing. For this example, we use the Febrl datasets 4A and 4B. These datasets can be loaded with the function `load_febrl4`.

```
In [2]: import recordlinkage
        from recordlinkage.datasets import load_febrl4
```

The datasets are loaded with the following code. The returned datasets are of type `pandas.DataFrame`. This makes it easy to manipulate the data if desired. For details about data manipulation with `pandas`, see their comprehensive documentation http://pandas.pydata.org/.

```
In [3]: dfA, dfB = load_febrl4()

        dfA
```

```
Out[3]: given_name    surname street_number            address_1  \
        rec_id
        rec-1070-org   michaela    neumann             8        stanley street
        rec-1016-org   courtney    painter            12     pinkerton circuit
        rec-4405-org    charles      green            38   salkauskas crescent
        rec-1288-org    vanessa       parr           905        macquoid place
        rec-3585-org    mikayla   malloney            37         randwick road
        ...                 ...        ...           ...                   ...
        rec-2153-org    annabel   grierson            97    mclachlan crescent
        rec-1604-org     sienna   musolino            22       smeaton circuit
        rec-1003-org    bradley   matthews             2          jondol place
        rec-4883-org     brodee       egan            88           axon street
        rec-66-org        koula  houweling             3        mileham street
```

```
                        address_2            suburb postcode state  \
rec_id
rec-1070-org                miami     winston hills     4223   nsw
rec-1016-org           bega flats         richlands     4560   vic
rec-4405-org                 kela             dapto     4566   nsw
rec-1288-org    broadbridge manor     south grafton     2135    sa
rec-3585-org              avalind  hoppers crossing     4552   vic
...                           ...               ...      ...   ...
rec-2153-org        lantana lodge            broome     2480   nsw
rec-1604-org              pangani          mckinnon     2700   nsw
rec-1003-org         horseshoe ck       jacobs well     7018    sa
rec-4883-org          greenslopes          wamberal     2067   qld
rec-66-org     old airdmillan road      williamstown     2350   nsw

             date_of_birth soc_sec_id
rec_id
rec-1070-org      19151111    5304218
rec-1016-org      19161214    4066625
rec-4405-org      19480930    4365168
rec-1288-org      19951119    9239102
rec-3585-org      19860208    7207688
...                    ...        ...
rec-2153-org      19840224    7676186
rec-1604-org      19890525    4971506
rec-1003-org      19481122    8927667
rec-4883-org      19121113    6039042
rec-66-org        19440718    6375537

[5000 rows x 10 columns]
```

3.2 Make record pairs

It is very intuitive to compare each record in DataFrame `dfA` with all records of DataFrame `dfB`. In fact, we want to make record pairs. Each record pair should contain one record of `dfA` and one record of `dfB`. This process of making record pairs is also called 'indexing'. With the `recordlinkage` module, indexing is easy. First, load the `FullIndex` class. This class makes a full index on a `.index(...)` call. In case of deduplication of a single dataframe, one dataframe is sufficient as argument.

```
In [4]: indexer = recordlinkage.FullIndex()
        pairs = indexer.index(dfA, dfB)
```

```
WARNING:recordlinkage:The number of record pairs is large. Consider a different indexation algorithm
```

With the method `index`, all possible (and unique) record pairs are made. The method returns a `pandas.MultiIndex`. The number of pairs is equal to the number of records in `dfA` times the number of records in `dfB`.

```
In [5]: print (len(dfA), len(dfB), len(pairs))
```

```
5000 5000 25000000
```

Many of these record pairs do not belong to the same person. In case of one-to-one matching, the number of matches should be no more than the number of records in the smallest dataframe. In case of full indexing, `min(len(dfA), len(N_dfB))` is much smaller than `len(pairs)`. The `recordlinkage` module has some more advanced indexing methods to reduce the number of record pairs. Obvious non-matches are left out of the index. Note that if a matching record pair is not included in the index, it can not be matched anymore.

One of the most well known indexing methods is named *blocking*. This method includes only record pairs that are identical on one or more stored attributes of the person (or entity in general). The blocking method can be used in the `recordlinkage` module.

```
In [6]: indexer = recordlinkage.BlockIndex(on='given_name')
        pairs = indexer.index(dfA, dfB)

        print (len(pairs))
```

```
77249
```

The argument 'given_name' is the blocking variable. This variable has to be the name of a column in `dfA` and `dfB`. It is possible to parse a list of columns names to block on multiple variables. Blocking on multiple variables will reduce the number of record pairs even further.

Another implemented indexing method is *Sorted Neighbourhood Indexing* (`recordlinkage.SortedNeighbourhoodIndex`). This method is very useful when there are many misspellings in the string were used for indexing. In fact, sorted neighbourhood indexing is a generalisation of blocking. See the documentation for details about sorted neighbourd indexing.

3.3 Compare records

Each record pair is a candidate match. To classify the candidate record pairs into matches and non-matches, compare the records on all attributes both records have in common. The `recordlinkage` module has a class named `Compare`. This class is used to compare the records. The following code shows how to compare attributes.

```
In [7]: # This cell can take some time to compute.
        compare_cl = recordlinkage.Compare()

        compare_cl.exact('given_name', 'given_name', label='given_name')
        compare_cl.string('surname', 'surname', method='jarowinkler', threshold=0.85, label='surname
        compare_cl.exact('date_of_birth', 'date_of_birth', label='date_of_birth')
        compare_cl.exact('suburb', 'suburb', label='suburb')
        compare_cl.exact('state', 'state', label='state')
        compare_cl.string('address_1', 'address_1', threshold=0.85, label='address_1')

        features = compare_cl.compute(pairs, dfA, dfB)
```

The comparing of record pairs starts when the `compute` method is called. All attribute comparisons are stored in a DataFrame with horizontally the features and vertically the record pairs.

```
In [8]: features
```

```
Out[8]: given_name  surname  date_of_birth  suburb  \
        rec_id       rec_id
        rec-1070-org rec-3024-dup-0          1      0.0              0       0
                     rec-2371-dup-0          1      0.0              0       0
                     rec-4652-dup-0          1      0.0              0       0
                     rec-4795-dup-0          1      0.0              0       0
                     rec-1314-dup-0          1      0.0              0       0
        ...                                ...      ...            ...     ...
        rec-4528-org rec-4528-dup-0          1      1.0              1       1
        rec-4887-org rec-4887-dup-0          1      1.0              1       0
        rec-4350-org rec-4350-dup-0          1      1.0              1       1
        rec-4569-org rec-4569-dup-0          1      1.0              1       1
        rec-3125-org rec-3125-dup-0          1      1.0              1       0

                                     state  address_1
        rec_id       rec_id
        rec-1070-org rec-3024-dup-0      1        0.0
                     rec-2371-dup-0      0        0.0
                     rec-4652-dup-0      0        0.0
                     rec-4795-dup-0      1        0.0
```

```
             rec-1314-dup-0      1        0.0
...                            ...        ...
rec-4528-org rec-4528-dup-0      1        1.0
rec-4887-org rec-4887-dup-0      1        1.0
rec-4350-org rec-4350-dup-0      1        1.0
rec-4569-org rec-4569-dup-0      1        0.0
rec-3125-org rec-3125-dup-0      1        1.0

[77249 rows x 6 columns]
```

```
In [9]: features.describe()
```

```
Out[9]: given_name      surname  date_of_birth       suburb         state  \
        count     77249.0  77249.00000    77249.00000  77249.00000  77249.00000
        mean          1.0      0.04443        0.03793      0.03226      0.24877
        std           0.0      0.20604        0.19103      0.17669      0.43230
        min           1.0      0.00000        0.00000      0.00000      0.00000
        25%           1.0      0.00000        0.00000      0.00000      0.00000
        50%           1.0      0.00000        0.00000      0.00000      0.00000
        75%           1.0      0.00000        0.00000      0.00000      0.00000
        max           1.0      1.00000        1.00000      1.00000      1.00000

                 address_1
        count  77249.00000
        mean       0.03670
        std        0.18802
        min        0.00000
        25%        0.00000
        50%        0.00000
        75%        0.00000
        max        1.00000
```

The last step is to decide which records belong to the same person. In this example, we keep it simple:

```
In [10]: # Sum the comparison results.
         features.sum(axis=1).value_counts().sort_index(ascending=False)
```

```
Out[10]: 6.0     1566
         5.0     1332
         4.0      343
         3.0      146
         2.0    16427
         1.0    57435
         dtype: int64
```

```
In [11]: features[features.sum(axis=1) > 3]
```

```
Out[11]: given_name  surname  date_of_birth  suburb  \
         rec_id       rec_id
         rec-2371-org rec-2371-dup-0          1      1.0              1       1
         rec-3024-org rec-3024-dup-0          1      1.0              1       0
         rec-4652-org rec-4652-dup-0          1      1.0              1       0
         rec-4795-org rec-4795-dup-0          1      1.0              1       1
         rec-1016-org rec-1016-dup-0          1      1.0              1       1
         ...                                ...      ...            ...     ...
         rec-4528-org rec-4528-dup-0          1      1.0              1       1
         rec-4887-org rec-4887-dup-0          1      1.0              1       0
         rec-4350-org rec-4350-dup-0          1      1.0              1       1
         rec-4569-org rec-4569-dup-0          1      1.0              1       1
         rec-3125-org rec-3125-dup-0          1      1.0              1       0

                                     state  address_1
```

```
rec_id       rec_id
rec-2371-org rec-2371-dup-0      1        1.0
rec-3024-org rec-3024-dup-0      1        0.0
rec-4652-org rec-4652-dup-0      1        1.0
rec-4795-org rec-4795-dup-0      1        1.0
rec-1016-org rec-1016-dup-0      0        1.0
...                            ...        ...
rec-4528-org rec-4528-dup-0      1        1.0
rec-4887-org rec-4887-dup-0      1        1.0
rec-4350-org rec-4350-dup-0      1        1.0
rec-4569-org rec-4569-dup-0      1        0.0
rec-3125-org rec-3125-dup-0      1        1.0

[3241 rows x 6 columns]
```

3.4 Full code

```
In [12]: import recordlinkage
         from recordlinkage.datasets import load_febrl4

         dfA, dfB = load_febrl4()

         # Indexation step
         indexer = recordlinkage.BlockIndex(on='given_name')
         pairs = indexer.index(dfA, dfB)

         # Comparison step
         compare_cl = recordlinkage.Compare()

         compare_cl.exact('given_name', 'given_name', label='given_name')
         compare_cl.string('surname', 'surname', method='jarowinkler', threshold=0.85, label='surname
         compare_cl.exact('date_of_birth', 'date_of_birth', label='date_of_birth')
         compare_cl.exact('suburb', 'suburb', label='suburb')
         compare_cl.exact('state', 'state', label='state')
         compare_cl.string('address_1', 'address_1', threshold=0.85, label='address_1')

         features = compare_cl.compute(pairs, dfA, dfB)

         # Classification step
         matches = features[features.sum(axis=1) > 3]
         print(len(matches))
```

```
3241
```

CHAPTER 4

Data deduplication

4.1 Introduction

This example shows how to find records in datasets belonging to the same entity. In our case,we try to deduplicate a dataset with records of persons. We will try to link within the dataset based on attributes like first name, surname, sex, date of birth, place and address. The data used in this example is part of Febrl and is fictitious.

First, start with importing the `recordlinkage` module. The submodule `recordlinkage.datasets` contains several datasets that can be used for testing. For this example, we use the Febrl dataset 1. This dataset contains 1000 records of which 500 original and 500 duplicates, with exactly one duplicate per original record. This dataset can be loaded with the function `load_febrl1`.

```
In [1]: import recordlinkage
        from recordlinkage.datasets import load_febrl1
```

The dataset is loaded with the following code. The returned datasets are of type `pandas.DataFrame`. This makes it easy to manipulate the data if desired. For details about data manipulation with `pandas`, see their comprehensive documentation http://pandas.pydata.org/.

```
In [2]: dfA = load_febrl1()

        dfA.head()

Out[2]: given_name     surname street_number          address_1  \
        rec_id
        rec-223-org          NaN      waller             6   tullaroop street
        rec-122-org      lachlan       berry            69      giblin street
        rec-373-org       deakin  sondergeld            48  goldfinch circuit
        rec-10-dup-0       kayla  harrington           NaN     maltby circuit
        rec-227-org         luke      purdon            23       ramsay place

                      address_2      suburb postcode state date_of_birth soc_sec_id
        rec_id
        rec-223-org    willaroo    st james     4011    wa      19081209    6988048
        rec-122-org   killarney     bittern     4814   qld      19990219    7364009
        rec-373-org     kooltuo  canterbury     2776   vic      19600210    2635962
        rec-10-dup-0     coaling     coolaroo      3465    nsw       19150612     9004242
        rec-227-org       mirani      garbutt      2260    vic       19831024     8099933
```

4.2 Make record pairs

It is very intuitive to start with comparing each record in DataFrame `dfA` with all other records in DataFrame `dfA`. In fact, we want to make record pairs. Each record pair should contain two different records of DataFrame `dfA`. This process of making record pairs is also called 'indexing'. With the `recordlinkage` module, indexing is easy. First, load the `FullIndex` class. This class makes a full index on a `.index(...)` call. In case of deduplication of a single dataframe, one dataframe is sufficient as input argument.

```
In [3]: indexer = recordlinkage.FullIndex()
        pairs = indexer.index(dfA)
```

With the method `index`, all possible (and unique) record pairs are made. The method returns a `pandas.MultiIndex`. The number of pairs is equal to the number of records in `dfA` times the number of records in `dfB`.

```
In [4]: print (len(dfA), len(pairs))
        # (1000*1000-1000)/2 = 499500
```

```
1000 499500
```

Many of these record pairs do not belong to the same person. The `recordlinkage` toolkit has some more advanced indexing methods to reduce the number of record pairs. Obvious non-matches are left out of the index. Note that if a matching record pair is not included in the index, it can not be matched anymore.

One of the most well known indexing methods is named *blocking*. This method includes only record pairs that are identical on one or more stored attributes of the person (or entity in general). The blocking method can be used in the `recordlinkage` module.

```
In [5]: indexer = recordlinkage.BlockIndex(on='given_name')
        pairs = indexer.index(dfA)

        print (len(pairs))
```

```
2082
```

The argument 'given_name' is the blocking variable. This variable has to be the name of a column in `dfA` and `dfB`. It is possible to parse a list of columns names to block on multiple variables. Blocking on multiple variables will reduce the number of record pairs even further.

Another implemented indexing method is *Sorted Neighbourhood Indexing* (`Pairs.sortedneighbourhood`). This method is very useful when there are many misspellings in the string were used for indexing. In fact, sorted neighbourhood indexing is a generalisation of blocking. See the documentation for details about sorted neighbourd indexing.

4.3 Compare records

Each record pair is a candidate match. To classify the candidate record pairs into matches and non-matches, compare the records on all attributes both records have in common. The `recordlinkage` module has a class named `Compare`. This class is used to compare the records. The following code shows how to compare attributes.

```
In [6]: # This cell can take some time to compute.
        compare_cl = recordlinkage.Compare()

        compare_cl.exact('given_name', 'given_name', label='given_name')
        compare_cl.string('surname', 'surname', method='jarowinkler', threshold=0.85, label='surname
        compare_cl.exact('date_of_birth', 'date_of_birth', label='date_of_birth')
        compare_cl.exact('suburb', 'suburb', label='suburb')
        compare_cl.exact('state', 'state', label='state')
        compare_cl.string('address_1', 'address_1', threshold=0.85, label='address_1')

        features = compare_cl.compute(pairs, dfA)
```

The comparing of record pairs starts when the `compute` method is called. All attribute comparisons are stored in a DataFrame with horizontally the features and vertically the record pairs. The first 10 comparison vectors are:

```
In [7]: features.head(10)
```

```
Out[7]: given_name  surname  date_of_birth  suburb  \
        rec_id        rec_id
        rec-122-org   rec-183-dup-0          1      0.0              0        0
                      rec-248-org            1      0.0              0        0
                      rec-469-org            1      0.0              0        0
                      rec-74-org             1      0.0              0        0
                      rec-183-org            1      0.0              0        0
                      rec-360-dup-0          1      0.0              0        0
                      rec-248-dup-0          1      0.0              0        0
                      rec-469-dup-0          1      0.0              0        0
        rec-183-dup-0 rec-248-org            1      0.0              0        0
                      rec-469-org            1      0.0              0        0

                                    state  address_1
        rec_id        rec_id
        rec-122-org   rec-183-dup-0      0        0.0
                      rec-248-org        1        0.0
                      rec-469-org        0        0.0
                      rec-74-org         0        0.0
                      rec-183-org        0        0.0
                      rec-360-dup-0      0        0.0
                      rec-248-dup-0      1        0.0
                      rec-469-dup-0      0        0.0
        rec-183-dup-0 rec-248-org        0        0.0
                      rec-469-org        1        0.0
```

```
In [8]: features.describe()
```

```
Out[8]: given_name      surname  date_of_birth       suburb        state  \
        count      2082.0  2082.000000    2082.000000  2082.000000  2082.000000
        mean          1.0     0.144092       0.139289     0.108549     0.327089
        std           0.0     0.351268       0.346331     0.311148     0.469263
        min           1.0     0.000000       0.000000     0.000000     0.000000
        25%           1.0     0.000000       0.000000     0.000000     0.000000
        50%           1.0     0.000000       0.000000     0.000000     0.000000
        75%           1.0     0.000000       0.000000     0.000000     1.000000
        max           1.0     1.000000       1.000000     1.000000     1.000000

                 address_1
        count  2082.000000
        mean      0.133045
        std       0.339705
        min       0.000000
        25%       0.000000
        50%       0.000000
        75%       0.000000
        max       1.000000
```

The last step is to decide which records belong to the same person. In this example, we keep it simple:

```
In [9]: # Sum the comparison results.
        features.sum(axis=1).value_counts().sort_index(ascending=False)
```

```
Out[9]: 6.0     142
        5.0     145
        4.0      30
        3.0       9
        2.0     376
        1.0    1380
        dtype: int64
```

```
In [10]: matches = features[features.sum(axis=1) > 3]

         print(len(matches))
         matches.head(10)
```

```
317
```

```
Out[10]: given_name  surname  date_of_birth  suburb  state  \
         rec_id        rec_id
         rec-183-dup-0 rec-183-org          1      1.0             1       1      1
         rec-122-dup-0 rec-122-org          1      1.0             1       1      1
         rec-248-dup-0 rec-248-org          1      1.0             1       1      1
         rec-373-dup-0 rec-373-org          1      1.0             1       1      1
         rec-10-dup-0  rec-10-org           1      1.0             1       1      1
         rec-342-dup-0 rec-342-org          1      1.0             0       1      1
         rec-330-dup-0 rec-330-org          1      0.0             1       1      1
         rec-397-dup-0 rec-397-org          1      1.0             1       1      1
         rec-472-dup-0 rec-472-org          1      1.0             1       1      1
         rec-190-dup-0 rec-190-org          1      1.0             0       1      1

                                    address_1
         rec_id        rec_id
         rec-183-dup-0 rec-183-org        1.0
         rec-122-dup-0 rec-122-org        1.0
         rec-248-dup-0 rec-248-org        1.0
         rec-373-dup-0 rec-373-org        1.0
         rec-10-dup-0  rec-10-org         1.0
         rec-342-dup-0 rec-342-org        1.0
         rec-330-dup-0 rec-330-org        0.0
         rec-397-dup-0 rec-397-org        0.0
         rec-472-dup-0 rec-472-org        0.0
         rec-190-dup-0 rec-190-org        1.0
```

4.4 Full code

```
In [11]: import recordlinkage
         from recordlinkage.datasets import load_febrl1

         dfA = load_febrl1()

         # Indexation step
         pcl = recordlinkage.BlockIndex(on='given_name')
         pairs = pcl.index(dfA)

         # Comparison step
         compare_cl = recordlinkage.Compare()
```

```
compare_cl.exact('given_name', 'given_name', label='given_name')
compare_cl.string('surname', 'surname', method='jarowinkler', threshold=0.85, label='surname
compare_cl.exact('date_of_birth', 'date_of_birth', label='date_of_birth')
compare_cl.exact('suburb', 'suburb', label='suburb')
compare_cl.exact('state', 'state', label='state')
compare_cl.string('address_1', 'address_1', threshold=0.85, label='address_1')

features = compare_cl.compute(pairs, dfA)

# Classification step
matches = features[features.sum(axis=1) > 3]
print(len(matches))
```

```
317
```

CHAPTER 5

Record pair classification

In the context of record linkage, classification refers to the process of dividing record pairs into matches and non-matches (distinct pairs). There are dozens of classification algorithms for record linkage. Roughly speaking, classification algorithms fall into two groups:

- **supervised learning algorithms** - These algorithms make use of trainings data. If you do have trainings data, then you can use supervised learning algorithms. Most supervised learning algorithms offer good accuracy and reliability. Examples of supervised learning algorithms in the *Python Record Linkage Toolkit* are *Logistic Regression*, *Naive Bayes* and *Support Vector Machines*.
- **unsupervised learning algorithms** - These algorithms do not need training data. The *Python Record Linkage Toolkit* supports *K-means clustering* and an *Expectation/Conditional Maximisation* classifier.

First things first

The examples below make use of the Krebs register (German for cancer registry) dataset. The Krebs register dataset contains comparison vectors of a large set of record pairs. For each record pair, it is known if the records represent the same person (match) or not (non-match). This was done with a massive clerical review. First, import the recordlinkage module and load the Krebs register data. The dataset contains 5749132 compared record pairs and has the following variables: first name, last name, sex, birthday, birth month, birth year and zip code. The Krebs register contains `len(krebs_match) == 20931` matching record pairs.

```
In [2]: import recordlinkage as rl
        from recordlinkage.datasets import load_krebsregister

        krebs_data, krebs_match = load_krebsregister(missing_values=0)
        krebs_data
```

```
Out[2]: cmp_firstname1  cmp_firstname2  cmp_lastname1  cmp_lastname2  \
        id1   id2
        22161 38467         1.00000             0.0        0.14286            0.0
        38713 75352         0.00000             0.0        0.57143            0.0
        13699 32825         0.16667             0.0        0.00000            0.0
        22709 37682         0.28571             0.0        1.00000            0.0
        2342  69060         0.25000             0.0        0.12500            0.0
        ...                     ...             ...            ...            ...
        52124 53629         1.00000             0.0        0.28571            0.0
```

```
30007 76846         0.75000             0.0        0.00000             0.0
50546 59461         0.75000             0.0        0.00000             0.0
43175 62151         1.00000             0.0        0.11111             0.0
11651 57925         1.00000             0.0        0.00000             0.0

             cmp_sex  cmp_birthday  cmp_birthmonth  cmp_birthyear  cmp_zipcode
id1   id2
22161 38467        1           0.0             1.0            0.0          0.0
38713 75352        1           0.0             0.0            0.0          0.0
13699 32825        0           1.0             1.0            1.0          0.0
22709 37682        1           0.0             0.0            0.0          0.0
2342  69060        1           1.0             1.0            1.0          0.0
...              ...           ...             ...            ...          ...
52124 53629        1           0.0             0.0            1.0          0.0
30007 76846        1           1.0             0.0            0.0          0.0
50546 59461        1           0.0             1.0            0.0          0.0
43175 62151        1           0.0             1.0            0.0          0.0
11651 57925        1           0.0             0.0            1.0          0.0

[5749132 rows x 9 columns]
```

Most classifiers can not handle comparison vectors with missing values. To prevent issues with the classification algorithms, we convert the missing values into disagreeing comparisons (using argument missing_values=0). This approach for handling missing values is widely used in record linkage applications.

```
In [3]: krebs_data.describe().T
```

```
Out[3]: count      mean       std  min       25%       50%       75%  \
        cmp_firstname1  5749132.0  0.71278  0.38884  0.0  0.28571  1.00000  1.00000
        cmp_firstname2  5749132.0  0.01623  0.12520  0.0  0.00000  0.00000  0.00000
        cmp_lastname1   5749132.0  0.31563  0.33423  0.0  0.10000  0.18182  0.42857
        cmp_lastname2   5749132.0  0.00014  0.01008  0.0  0.00000  0.00000  0.00000
        cmp_sex         5749132.0  0.95500  0.20730  0.0  1.00000  1.00000  1.00000
        cmp_birthday    5749132.0  0.22443  0.41721  0.0  0.00000  0.00000  0.00000
        cmp_birthmonth  5749132.0  0.48879  0.49987  0.0  0.00000  0.00000  1.00000
        cmp_birthyear   5749132.0  0.22272  0.41607  0.0  0.00000  0.00000  0.00000
        cmp_zipcode     5749132.0  0.00552  0.07407  0.0  0.00000  0.00000  0.00000

                        max
        cmp_firstname1  1.0
        cmp_firstname2  1.0
        cmp_lastname1   1.0
        cmp_lastname2   1.0
        cmp_sex         1.0
        cmp_birthday    1.0
        cmp_birthmonth  1.0
        cmp_birthyear   1.0
        cmp_zipcode     1.0
```

5.1 Supervised learning

As described before, supervised learning algorithms do need training data. Training data is data for which the true match status is known for each comparison vector. In the example in this section, we consider that the true match status of the first 5000 record pairs of the Krebs register data is known.

```
In [4]: golden_pairs = krebs_data[0:5000]
        golden_matches_index = golden_pairs.index & krebs_match # 2093 matching pairs
```

5.1.1 Logistic regression

The `recordlinkage.LogisticRegressionClassifier` classifier is an application of the logistic regression model. This supervised learning method is one of the oldest classification algorithms used in record linkage. In situations with enough training data, the algorithm gives relatively good results.

```
In [5]: # Initialize the classifier
        logreg = rl.LogisticRegressionClassifier()

        # Train the classifier
        logreg.learn(golden_pairs, golden_matches_index)
        print ("Intercept: ", logreg.intercept)
        print ("Coefficients: ", logreg.coefficients)
```

```
Intercept:  -6.2980435710064135
Coefficients:  [ 4.90452843e-01   1.21640484e-01   2.15040485e+00  -2.84818101e-03
  -1.79712465e+00   9.61085558e-01   6.72610441e-02   1.03408608e+00
   4.30556110e+00]
```

```
In [6]: # Predict the match status for all record pairs
        result_logreg = logreg.predict(krebs_data)

        len(result_logreg)
```

```
Out[6]: 20150
```

```
In [7]: conf_logreg = rl.confusion_matrix(krebs_match, result_logreg, len(krebs_data))
        conf_logreg
```

```
Out[7]: array([[  19884,     1047],
               [    266, 5727935]])
```

```
In [8]: # The F-score for this prediction is
        rl.fscore(conf_logreg)
```

```
Out[8]: 0.96804
```

The predicted number of matches is not much more than the 20931 true matches. The result was achieved with a small training dataset of 5000 record pairs.

In (older) literature, record linkage procedures are often divided in **deterministic record linkage** and **probabilistic record linkage**. The Logistic Regression Classifier belongs to deterministic record linkage methods. Each feature/variable has a certain importance (named weight). The weight is multiplied with the comparison/similarity vector. If the total sum exceeds a certain threshold, it as considered to be a match.

```
In [9]: intercept = -9
        coefficients = [2.0, 1.0, 3.0, 1.0, 1.0, 1.0, 1.0, 2.0, 3.0]

        logreg = rl.LogisticRegressionClassifier(coefficients, intercept)

        # predict without calling LogisticRegressionClassifier.learn
        matches = logreg.predict(krebs_data)
        print (len(matches))
```

```
21303
```

```
In [10]: conf_logreg = rl.confusion_matrix(krebs_match, matches, len(krebs_data))
         conf_logreg

         # The F-score for this classification is
         rl.fscore(conf_logreg)
```

```
Out[10]: 0.98769
```

For the given coefficients, the F-score is better than the situation without trainings data. Surprising? No (use more trainings data and the result will improve)

5.1.2 Naive Bayes

In contrast to the logistic regression classifier, the Naive Bayes classifier is a probabilistic classifier. The probabilistic record linkage framework by Fellegi and Sunter (1969) is the most well-known probabilistic classification method for record linkage. Later, it was proved that the Fellegi and Sunter method is mathematically equivalent to the Naive Bayes method in case of assuming independence between comparison variables.

```
In [11]: # Train the classifier
         nb = rl.NaiveBayesClassifier()
         nb.learn(golden_pairs, golden_matches_index)

         # Predict the match status for all record pairs
         result_nb = nb.predict(krebs_data)

         len(result_nb)

Out[11]: 20345

In [12]: conf_nb = rl.confusion_matrix(krebs_match, result_nb, len(krebs_data))
         conf_nb

Out[12]: array([[  20023,     908],
                [    322, 5727879]])

In [13]: # The F-score for this classification is
         rl.fscore(conf_nb)

Out[13]: 0.97020
```

5.1.3 Support Vector Machines

Support Vector Machines (SVM) have become increasingly popular in record linkage. The algorithm performs well there is only a small amount of training data available. The implementation of SVM in the Python Record Linkage Toolkit is a linear SVM algorithm.

```
In [14]: # Train the classifier
         svm = rl.SVMClassifier()
         svm.learn(golden_pairs, golden_matches_index)

         # Predict the match status for all record pairs
         result_svm = svm.predict(krebs_data)

         len(result_svm)

Out[14]: 20839

In [15]: conf_svm = rl.confusion_matrix(krebs_match, result_svm, len(krebs_data))
         conf_svm

Out[15]: array([[  20825,     106],
                [     14, 5728187]])

In [16]: # The F-score for this classification is
         rl.fscore(conf_svm)

Out[16]: 0.99713
```

5.2 Unsupervised learning

In situations without training data, unsupervised learning can be a solution for record linkage problems. In this section, we discuss two unsupervised learning methods. One algorithm is K-means clustering, and the other algorithm is an implementation of the Expectation-Maximisation algorithm. Most of the time, unsupervised learning algorithms take more computational time because of the iterative structure in these algorithms.

5.2.1 K-means clustering

The K-means clustering algorithm is well-known and widely used in big data analysis. The K-means classifier in the Python Record Linkage Toolkit package is configured in such a way that it can be used for linking records. For more info about the K-means clustering see Wikipedia.

```
In [17]: kmeans = rl.KMeansClassifier()
         result_kmeans = kmeans.learn(krebs_data)

         # The predicted number of matches
         len(result_kmeans)
```

```
Out[17]: 371525
```

The classifier is now trained and the comparison vectors are classified.

```
In [18]: cm_kmeans = rl.confusion_matrix(krebs_match, result_kmeans, len(krebs_data))

         rl.fscore(cm_kmeans)
```

```
Out[18]: 0.10598
```

5.2.2 Expectation/Conditional Maximization Algorithm

The ECM-algorithm is an Expectation-Maximisation algorithm with some additional constraints. This algorithm is closely related to the Naive Bayes algorithm. The ECM algorithm is also closely related to estimating the parameters in the Fellegi and Sunter (1969) framework. The algorithms assume that the attributes are independent of each other. The Naive Bayes algorithm uses the same principles.

```
In [19]: # Train the classifier
         ecm = rl.ECMClassifier()
         result_ecm = ecm.learn((krebs_data > 0.8).astype(int))

         len(result_ecm)
```

```
Out[19]: 19817
```

```
In [20]: conf_ecm = rl.confusion_matrix(krebs_match, result_ecm, len(krebs_data))
         conf_ecm
```

```
Out[20]: array([[  19813,    1118],
                [      4, 5728197]])
```

```
In [21]: # The F-score for this classification is
         rl.fscore(conf_ecm)
```

```
Out[21]: 0.97246
```

CHAPTER 6

Custom algorithms

The Python Record Linkage Toolkit contains several built-in algorithms for making record pairs and comparing record pairs. Sometimes, these built-in algorithms do not fit your needs. With the Python Record Linkage Toolkit, it is easy to use other algorithms. This section describes how to implement custom algorithms for the making and comparing record pairs. When you think your algorithm might help others, consider sharing it!

For run the examples below, import *pandas*, *recordlinkage* and the two datasets belonging to sample dataset FEBRL4.

```
In [2]: import pandas

        import recordlinkage
        from recordlinkage.datasets import load_febrl4

        df_a, df_b = load_febrl4()
```

```
In [3]: df_a
```

```
Out[3]: given_name    surname street_number            address_1  \
        rec_id
        rec-1070-org   michaela    neumann             8       stanley street
        rec-1016-org   courtney    painter            12    pinkerton circuit
        rec-4405-org    charles      green            38  salkauskas crescent
        rec-1288-org    vanessa       parr           905        macquoid place
        rec-3585-org    mikayla   malloney            37         randwick road
        ...                 ...        ...           ...                  ...
        rec-2153-org    annabel   grierson            97    mclachlan crescent
        rec-1604-org     sienna   musolino            22       smeaton circuit
        rec-1003-org    bradley   matthews             2          jondol place
        rec-4883-org     brodee       egan            88          axon street
        rec-66-org        koula  houweling             3       mileham street

                                address_2          suburb postcode state  \
        rec_id
        rec-1070-org                miami   winston hills     4223   nsw
        rec-1016-org           bega flats       richlands     4560   vic
        rec-4405-org                 kela           dapto     4566   nsw
        rec-1288-org    broadbridge manor   south grafton     2135    sa
```

```
rec-3585-org             avalind  hoppers crossing     4552   vic
...                          ...               ...      ...   ...
rec-2153-org        lantana lodge           broome     2480   nsw
rec-1604-org              pangani         mckinnon     2700   nsw
rec-1003-org         horseshoe ck     jacobs well      7018    sa
rec-4883-org          greenslopes         wamberal     2067   qld
rec-66-org     old airdmillan road     williamstown    2350   nsw

             date_of_birth soc_sec_id
rec_id
rec-1070-org      19151111    5304218
rec-1016-org      19161214    4066625
rec-4405-org      19480930    4365168
rec-1288-org      19951119    9239102
rec-3585-org      19860208    7207688
...                    ...        ...
rec-2153-org      19840224    7676186
rec-1604-org      19890525    4971506
rec-1003-org      19481122    8927667
rec-4883-org      19121113    6039042
rec-66-org        19440718    6375537

[5000 rows x 10 columns]
```

6.1 Custom index algorithms

The Python Record Linkage Toolkit contains multiple algorithms to pair records (index algorithms) such as full indexing, blocking and sorted neighbourhood indexing. This section explains how to make and implement a custom algorithm to make record pairs.

To create a custom algorithm, you have to make a subclass of the `recordlinkage.base.BaseIndexator`. In your subclass, you overwrite the `_link_index` method. This method accepts two pandas DataFrames as arguments. Based on these DataFrames, your method must create pairs and return them as a `pandas.MultiIndex` in which the MultiIndex names are the index names of DataFrame A and DataFrame B respectively.

The algorithm for linking data frames can be used for finding duplicates as well. In this situation, DataFrame B is a copy of DataFrame A. The `Pairs` class removes pairs like `(record_i, record_i)` and one of the following `(record_i, record_j)` `(record_j, record_i)` under the hood. As result of this, only unique combinations are returned. If you do have a specific algorithm for finding duplicates, then you can overwrite the `_dedup_index` method. This method accepts only one argument (DataFrame A) and the internal base class does not look for combinations like explained above.

Let's make an algorithm that pairs records of which the given name in both records starts with the letter 'W'.

```
In [4]: from recordlinkage.base import BaseIndexator

        class FirstLetterWIndex(BaseIndexator):
            """Custom class for indexing"""

            def _link_index(self, df_a, df_b):
                """Make pairs of all records where the given name start with the letter 'W'."""

                # Select records with names starting with a w.
                name_a_startswith_w = df_a[df_a['given_name'].str.startswith('w') == True]
                name_b_startswith_w = df_b[df_b['given_name'].str.startswith('w') == True]

                # Make a product of the two numpy arrays
                return pandas.MultiIndex.from_product(
                    [name_a_startswith_w.index.values, name_b_startswith_w.index.values],
                    names=[df_a.index.name, df_b.index.name]
                )
```

```
In [5]: indexer = FirstLetterWIndex()
        candidate_pairs = indexer.index(df_a, df_b)

        print ('Returns a', type(candidate_pairs).__name__)
        print ('Number of candidate record pairs starting with the letter w:', len(candidate_pairs))
```

```
Returns a MultiIndex
Number of candidate record pairs starting with the letter w: 6072
```

The custom index class below does not restrict the first letter to 'w', but the first letter is an argument (named `letter`). This letter can is initialized during the setup of the class.

```
In [6]: class FirstLetterIndex(BaseIndexator):
            """Custom class for indexing"""

            def __init__(self, letter):
                super(FirstLetterIndex, self).__init__()

                # the letter to save
                self.letter = letter

            def _link_index(self, df_a, df_b):
                """Make record pairs that agree on the first letter of the given name."""

                # Select records with names starting with a 'letter'.
                a_startswith_w = df_a[df_a['given_name'].str.startswith(self.letter) == True]
                b_startswith_w = df_b[df_b['given_name'].str.startswith(self.letter) == True]

                # Make a product of the two numpy arrays
                return pandas.MultiIndex.from_product(
                    [a_startswith_w.index.values, b_startswith_w.index.values],
                    names=[df_a.index.name, df_b.index.name]
                )
```

```
In [7]: indexer = FirstLetterIndex('w')
        candidate_pairs = indexer.index(df_a, df_b)

        print ('Number of record pairs (letter w):', len(candidate_pairs))
```

```
Number of record pairs (letter w): 6072
```

```
In [8]: for letter in 'wxa':

            indexer = FirstLetterIndex(letter)
            candidate_pairs = indexer.index(df_a, df_b)

            print ('Number of record pairs (letter %s):' % letter, len(candidate_pairs))
```

```
Number of record pairs (letter w): 6072
Number of record pairs (letter x): 132
Number of record pairs (letter a): 172431
```

6.2 Custom compare algorithms

The Python Record Linkage Toolkit holds algorithms to compare strings, numeric values and dates. These functions may not be sufficient for your record linkage. The internal framework of the toolkit makes it easy to implement custom algorithms to compare records. A custom function can be passed to the `compare_vectorized` method (see example below).

A custom algorithm is a function that accepts at least two arguments. The first argument is a `pandas.Series` with information on the variable in the first file and the second argument is a `pandas.Series` with information on the variable in the second file. The custom function has to return a `pandas.Series`, `numpy.array` or list with the similarity/comparison values.

The following code is a custom function to compare zipcodes. The function returns 1.0 for record pairs that agree on the zipcode and returns 0.0 for records that disagree on the zipcode. If the zipcodes disagree but the first two numbers are identical, then the function returns 0.5.

```
In [9]: def compare_zipcodes(s1, s2):
            """
            If the zipcodes in both records are identical, the similarity
            is 0. If the first two values agree and the last two don't, then
            the similarity is 0.5. Otherwise, the similarity is 0.
            """

            # check if the zipcode are identical (return 1 or 0)
            sim = (s1 == s2).astype(float)

            # check the first 2 numbers of the distinct comparisons
            sim[(sim == 0) & (s1.str[0:2] == s2.str[0:2])] = 0.5

            return sim
```

```
In [10]: # Make an index of record pairs
         pcl = recordlinkage.BlockIndex('given_name')
         candidate_pairs = pcl.index(df_a, df_b)

         comparer = recordlinkage.Compare()
         comparer.compare_vectorized(compare_zipcodes, 'postcode', 'postcode', label='sim_postcode')
         features = comparer.compute(candidate_pairs, df_a, df_b)

         features['sim_postcode'].value_counts()
```

```
Out[10]: 0.0    71229
         0.5     3166
         1.0     2854
         Name: sim_postcode, dtype: int64
```

As you can see, one can pass the labels of the columns as arguments. The first argument is a column label, or a list of column labels, found in the first DataFrame (`postcode` in this example). The second argument is a column label, or a list of column labels, found in the second DataFrame (also `postcode` in this example). The `recordlinkage.Compare` class selects the columns with the given labels before passing them to the custom algorithm/function. The `compare` method in the `recordlinkage.Compare` class passes additional (keyword) arguments to the custom function.

Warning: Do not change the order of the pairs in the MultiIndex.

6.2.1 Multicolumn comparisons

The Python Record Linkage Toolkit supports the comparison of more than two columns. This is especially useful in situations with multi-dimensional data (for example geographical coordinates) and situations where fields can be swapped.

The FEBRL4 dataset has two columns filled with address information (`address_1` and `address_2`). In a naive approach, one compares `address_1` of file A with `address_1` of file B and `address_2` of file A with `address_2` of file B. If the values for `address_1` and `address_2` are swapped during the record generating process, the naive approach considers the addresses to be distinct. In a more advanced approach, `address_1` of file A is compared with `address_1` and `address_2` of file B. Variable `address_2` of file A is compared with `address_1` and `address_2` of file B. This is done with the single function given below.

```
In [11]: def compare_addresses(s1_1, s1_2, s2_1, s2_2):
             """
             Compare addresses. Compare address_1 of file A with
             address_1 and address_2 of file B. The same for address_2
             of dataset 1.

             """

             return ((s1_1 == s2_1) | (s1_2 == s2_2) | (s1_1 == s2_2) | (s1_2 == s2_1)).astype(float
```

```
In [12]: comparer = recordlinkage.Compare()

         # naive
         comparer.exact('address_1', 'address_1', label='sim_address_1')
         comparer.exact('address_2', 'address_2', label='sim_address_2')

         # better
         comparer.compare_vectorized(
             compare_addresses,
             ('address_1', 'address_2'), ('address_1', 'address_2'),
             label='sim_address'
         )

         features = comparer.compute(candidate_pairs, df_a, df_b)

         features.mean()
```

```
Out[12]: sim_address_1    0.02488
         sim_address_2    0.02025
         sim_address      0.03566
         dtype: float64
```

CHAPTER 7

Performance

Performance plays an important role in record linkage. Record linkage problems scale quadratically with the size of the file(s). The number of record pairs can be enormous and so are the number of comparisons. Over the years, smart techniques are developed to reduce the number of record pairs. The *Python Record Linkage Toolkit* offers several effective techniques for making record pairs. Nevertheless, the Python Record Linkage Toolkit is **NOT** developed with speed in mind. **Understandability**, **usability** and **extensibility** are the main focus points in the development. This makes the toolkit very useful for linking small or medium sized files.

Okeee… There is not only bad news. The performance of many algorithms is good. Nevertheless, there are plenty of options to increase the performance of your record linkage implementation. Some methods are discussed in this topic.

Do you know more tricks? Let us know!

7.1 Indexation

7.1.1 Block on multiple columns

Blocking is an effective way to increase the performance of your record linkage. If the performance of your implementation is still poor, decrease the number of pairs by blocking on multiple variables. This implies that the record pair is agrees on two or more variables. In the following example, the record pairs agree on the given name **and** surname.

```
p = recordlinkage.BlockIndex(left_on=['first_name', 'surname'],
                             right_on=['name', 'surname'])
pairs = p.index(dfA, dfB)
```

You might exclude more links then desired. This can be solved by repeating the process with a different combinations of blocking variables. In the end, merge the links of the two, so called, index passes.

```
p_surname = recordlinkage.BlockIndex(left_on=['first_name', 'surname'],
                                     right_on=['name', 'surname'])
p_age = recordlinkage.BlockIndex(left_on=['first_name', 'age'],
                                 right_on=['name', 'age'])

```

```
pairs_surname = p_surname.index(dfA, dfB)
pairs_age = p_age.index(dfA, dfB)

# make a union of the candidate links of both classes
pairs_surname.union(pairs_age)
```

Note: Sorted Neighbourhood indexing supports, besides the sorted neighbourhood, additional blocking on variables.

7.1.2 Make record pairs

The structure of the Python Record Linkage Toolkit has a drawback for the performance. In the indexation step (the step in which record pairs are selected), only the index of both records is stored. The entire records are not stored. This results in less memory usage. The drawback is that the records need to be queried from the data.

7.2 Comparing

7.2.1 Compare only discriminating variables

Not all variables may be worth comparing in a record linkage. Some variables do not discriminate the links of the non-links or do have only minor effects. These variables can be excluded. Only discriminating and informative should be included.

7.2.2 Prevent string comparisons

String similarity measures and phonetic encodings are computationally expensive. Phonetic encoding takes place on the original data, while string simililatiry measures are applied on the record pairs. After phonetic encoding of the string variables, exact comparing can be used instead of computing the string similarity of all record pairs. If the number of candidate pairs is much larger than the number of records in both datasets together, then consider using phonetic encoding of string variables instead of string comparison.

7.2.3 String comparing

Comparing strings is computationally expensive. The Python Record Linkage Toolkit uses the package `jellyfish` for string comparisons. The package has two implementations, a C and a Python implementation. Ensure yourself of having the C-version installed (`import jellyfish.cjellyfish` should not raise an exception).

There can be a large difference in the performance of different string comparison algorithms. The Jaro and Jaro-Winkler methods are faster than the Levenshtein distance and much faster than the Damerau-Levenshtein distance.

7.2.4 Indexing with large files

Sometimes, the input files are very large. In that case, it can be hard to make an index without running out of memory in the indexing step or in the comparing step. `recordlinkage` has a method to deal with large files. It is fast, although is not primary developed to be fast. SQL databases may outperform this method. It is especially developed for the useability. The idea was to spllit the input files into small blocks. For each block the record pairs are computed. Then iterate over the blocks. Consider full indexing:

```
import recordlinkage
import numpy

cl = recordlinkage.FullIndex()

for dfB_subset in numpy.split(dfB):

    # a subset of record pairs
    pairs_subset = cl.index(dfA, dfB_subset)

    # Your analysis on pairs_subset here
```

CHAPTER 8

API Reference

This page contains the API reference of the Python Record Linkage Toolkit.

8.1 Preprocessing

Cleaning and preprocessing your data may increase your record linkage accuracy. The Python Record Linkage Toolkit contains several tools for data cleaning and preprocessing. Some of the tools included are: phonetic encoding algorithms and string cleaning tools. The tools are included in the submodule `preprocessing`. (Import example: `from recordlinkage.preprocessing import clean, phonenumbers`)

`clean`(*s*, *lowercase=True*, *replace_by_none='[^ \\-\\_A-Za-z0-9]+'*, *replace_by_whitespace='[\\-\\_]'*, *strip_accents=None*, *remove_brackets=True*, *encoding='utf-8'*, *decode_error='strict'*)

Clean string variables.

Clean strings in the Series by removing unwanted tokens, whitespace and brackets.

Parameters

- **`s`** (*`pandas.Series`*) – A Series to clean.
- **`lower`** (*`bool, optional`*) – Convert strings in the Series to lowercase. Default True.
- **`replace_by_none`** (*`str, optional`*) – The matches of this regular expression are replaced by ''.
- **`replace_by_whitespace`** (*`str, optional`*) – The matches of this regular expression are replaced by a whitespace.
- **`remove_brackets`** (*`bool, optional`*) – Remove all content between brackets and the brackets themselves. Default True.
- **`strip_accents`** (*`{'ascii', 'unicode', None}, optional`*) – Remove accents during the preprocessing step. 'ascii' is a fast method that only works on characters that have an direct ASCII mapping. 'unicode' is a slightly slower method that works on any characters. None (default) does nothing.

- **encoding** (`string`, *optional*) – If bytes are given, this encoding is used to decode. Default is 'utf-8'.
- **decode_error** ({*'strict'*, *'ignore'*, *'replace'*}, *optional*) – Instruction on what to do if a byte Series is given that contains characters not of the given *encoding*. By default, it is 'strict', meaning that a UnicodeDecodeError will be raised. Other values are 'ignore' and 'replace'.

Example

```
>>> import pandas
>>> from recordlinkage.preprocessing import clean
>>>
>>> name = ['Mary-ann', 'Bob :)', 'Angel', 'Bob (alias Billy)', None]
>>> s = pandas.Series(names)
>>> print(clean(s))
0    mary ann
1         bob
2       angel
3         bob
4         NaN
dtype: object
```

Returns *pandas.Series* – A cleaned Series of strings.

phonenumbers (*s*)
Clean phonenumbers by removing all non-numbers (except +).

Parameters **s** (*pandas.Series*) – A Series to clean.

Returns *pandas.Series* – A Series with cleaned phonenumbers.

value_occurence (*s*)
Count the number of times each value occurs.

This function returns the counts for each row, in contrast with pandas.value_counts.

Returns *pandas.Series* – A Series with value counts.

phonetic (*s*, *method*, *concat=True*, *encoding='utf-8'*, *decode_error='strict'*)
Convert names or strings into phonetic codes.

The implemented algorithms are soundex, nysiis, metaphone or match_rating.

Parameters

- **method** (`string`) – The algorithm that is used to phonetically encode the values. The possible options are "soundex", "nysiis", "metaphone" or "match rating".
- **concat** (*bool*, *optional*) – Remove whitespace before phonetic encoding.
- **encoding** (`string`, *optional*) – If bytes are given, this encoding is used to decode. Default is 'utf-8'.
- **decode_error** ({*'strict'*, *'ignore'*, *'replace'*}, *optional*) – Instruction on what to do if a byte Series is given that contains characters not of the given *encoding*. By default, it is 'strict', meaning that a UnicodeDecodeError will be raised. Other values are 'ignore' and 'replace'.:type method: str

Returns *pandas.Series* – A Series with phonetic encoded values.

Note: The 'soundex' and 'nysiis' algorithms use the package 'jellyfish'. It can be installed with pip (`pip install jellyfish`).

8.2 Indexing

Migrating from recordlinkage<=0.8.2 to recordlinkage>=0.9? *Click here*

The indexing module is used to make pairs of records. These pairs are called candidate links or candidate matches. There are several indexing algorithms available such as blocking and sorted neighborhood indexing. See the following references for background information about indexation.

`class FullIndex`

Class to generate a 'full' index.

A full index is an index with all possible combinations of record pairs. In case of linking, this indexation method generates the cartesian product of both DataFrame's. In case of deduplicating DataFrame A, this indexation method are the pairs defined by the upper triangular matrix of the A x A.

Note: This indexation method can be slow for large DataFrame's. The number of comparisons scales quadratic. Also, not all classifiers work well with large numbers of record pairs were most of the pairs are distinct.

`index`(*x*, *x_link=None*)

Make an index of record pairs.

Use a custom function to make record pairs of one or two dataframes. Each function should return a pandas.MultiIndex with record pairs.

Parameters

- **x** (*`pandas.DataFrame`*) – A pandas DataFrame. When *x_link* is None, the algorithm makes record pairs within the DataFrame. When *x_link* is not empty, the algorithm makes pairs between *x* and *x_link*.
- **x_link** (*`pandas.DataFrame, optional`*) – A second DataFrame to link with the DataFrame x.

Returns *pandas.MultiIndex* – A pandas.MultiIndex with record pairs. Each record pair contains the index labels of two records.

`class BlockIndex`(*on=None*, *left_on=None*, *right_on=None*)

Make candidate record pairs that agree on one or more variables.

Returns all record pairs that agree on the given variable(s). This method is known as *blocking*. Blocking is an effective way to make a subset of the record space (A * B).

Parameters

- **on** (*`label, optional`*) – A column name or a list of column names. These column(s) are used to block on. When linking two dataframes, the 'on' argument needs to be present in both dataframes.
- **left_on** (*`label, optional`*) – A column name or a list of column names of dataframe A. These columns are used to block on. This argument is ignored when argument 'on' is given.

- **right_on** (*label, optional*) – A column name or a list of column names of dataframe B. These columns are used to block on. This argument is ignored when argument 'on' is given.

Examples

In the following example, the record pairs are made for two historical datasets with census data. The datasets are named `census_data_1980` and `census_data_1990`.

```
>>> indexer = recordlinkage.BlockIndex(on='first_name')
>>> indexer.index(census_data_1980, census_data_1990)
```

index (*x, x_link=None*)

Make an index of record pairs.

Use a custom function to make record pairs of one or two dataframes. Each function should return a pandas.MultiIndex with record pairs.

Parameters

- **x** (*pandas.DataFrame*) – A pandas DataFrame. When *x_link* is None, the algorithm makes record pairs within the DataFrame. When *x_link* is not empty, the algorithm makes pairs between *x* and *x_link*.
- **x_link** (*pandas.DataFrame, optional*) – A second DataFrame to link with the DataFrame x.

Returns *pandas.MultiIndex* – A pandas.MultiIndex with record pairs. Each record pair contains the index labels of two records.

class SortedNeighbourhoodIndex (*on=None, left_on=None, right_on=None, window=3, sorting_key_values=None, block_on=[], block_left_on=[], block_right_on=[]*)

Make candidate record pairs with the SortedNeighbourhood algorithm.

This algorithm returns record pairs that agree on the sorting key, but also records pairs in their neighbourhood. A large window size results in more record pairs. A window size of 1 returns the blocking index.

The Sorted Neighbourhood Index method is a great method when there is relatively large amount of spelling mistakes. Blocking will fail in that situation because it excludes to many records on minor spelling mistakes.

Parameters

- **on** (*label, optional*) – The column name of the sorting key. When linking two dataframes, the 'on' argument needs to be present in both dataframes.
- **left_on** (*label, optional*) – The column name of the sorting key of the first/left dataframe. This argument is ignored when argument 'on' is not None.
- **right_on** (*label, optional*) – The column name of the sorting key of the second/right dataframe. This argument is ignored when argument 'on' is not None.
- **window** (*int, optional*) – The width of the window, default is 3
- **sorting_key_values** (*array, optional*) – A list of sorting key values (optional).
- **block_on** (*label*) – Additional columns to apply standard blocking on.
- **block_left_on** (*label*) – Additional columns in the left dataframe to apply standard blocking on.

- **block_right_on** (*label*) – Additional columns in the right dataframe to apply standard blocking on.

Examples

In the following example, the record pairs are made for two historical datasets with census data. The datasets are named `census_data_1980` and `census_data_1990`.

```
>>> indexer = recordlinkage.SortedNeighbourhoodIndex('first_name', window=9)
>>> indexer.index(census_data_1980, census_data_1990)
```

When the sorting key has different names in both dataframes:

```
>>> indexer = recordlinkage.SortedNeighbourhoodIndex(
        left_on='first_name', right_on='given_name', window=9
    )
>>> indexer.index(census_data_1980, census_data_1990)
```

index (*x*, *x_link=None*)

Make an index of record pairs.

Use a custom function to make record pairs of one or two dataframes. Each function should return a pandas.MultiIndex with record pairs.

Parameters

- **x** (*pandas.DataFrame*) – A pandas DataFrame. When *x_link* is None, the algorithm makes record pairs within the DataFrame. When *x_link* is not empty, the algorithm makes pairs between *x* and *x_link*.
- **x_link** (*pandas.DataFrame, optional*) – A second DataFrame to link with the DataFrame x.

Returns *pandas.MultiIndex* – A pandas.MultiIndex with record pairs. Each record pair contains the index labels of two records.

class RandomIndex (*n*, *replace=True*, *random_state=None*)

Class to generate random pairs of records.

This class returns random pairs of records with or without replacement. Use the random_state parameter to seed the algorithm and reproduce results. This way to make record pairs is useful for the training of unsupervised learning models for record linkage.

Parameters

- **n** (*int*) – The number of record pairs to return. In case replace=False, the integer n should be bounded by 0 < n <= n_max where n_max is the maximum number of pairs possible.
- **replace** (*bool, optional*) – Whether the sample of record pairs is with or without replacement. Default: True
- **random_state** (*int or numpy.random.RandomState, optional*) – Seed for the random number generator (if int), or numpy RandomState object.

index (*x*, *x_link=None*)

Make an index of record pairs.

Use a custom function to make record pairs of one or two dataframes. Each function should return a pandas.MultiIndex with record pairs.

Parameters

- **x** (*pandas.DataFrame*) – A pandas DataFrame. When *x_link* is None, the algorithm makes record pairs within the DataFrame. When *x_link* is not empty, the algorithm makes pairs between *x* and *x_link*.
- **x_link** (*pandas.DataFrame, optional*) – A second DataFrame to link with the DataFrame x.

Returns *pandas.MultiIndex* – A pandas.MultiIndex with record pairs. Each record pair contains the index labels of two records.

8.2.1 Migrating

Version 0.9 of the Python Record Linkage Toolkit uses a new indexing API. The new indexing API uses a different syntax. With the new API, each algorithm has it's own class. See the following example to migrate a blocking index:

Old (linking):

```
cl = recordlinkage.Pairs(df_a, df_b)
cl.block('given_name')
```

New (linking):

```
cl = recordlinkage.BlockIndex('given_name')
cl.index(df_a, df_b)
```

Old (deduplication):

```
cl = recordlinkage.Pairs(df_a)
cl.block('given_name')
```

New (deduplication):

```
cl = recordlinkage.BlockIndex('given_name')
cl.index(df_a)
```

8.3 Comparing

A set of informative, discriminating and independent features is important for a good classification of record pairs into matching and distinct pairs. The `recordlinkage.Compare` class and its methods can be used to compare records pairs. Several comparison methods are included such as string similarity measures, numerical measures and distance measures.

class Compare(*n_jobs=1*, *indexing_type='label'*)

Class to compare record pairs with efficiently.

Class to compare the attributes of candidate record pairs. The `Compare` class has methods like `string`, `exact` and `numeric` to initialise the comparing of the records. The `compute` method is used to start the actual comparing.

Parameters

- **n_jobs** (*integer, optional (default=1)*) – The number of jobs to run in parallel for comparing of record pairs. If -1, then the number of jobs is set to the number of cores.

- **indexing_type** (*string, optional (default='label')*) – The indexing type. The MultiIndex is used to index the DataFrame(s). This can be done with pandas `.loc` or with `.iloc`. Use the value 'label' to make use of `.loc` and 'position' to make use of `.iloc`. The value 'position' is only available when the MultiIndex consists of integers. The value 'position' is much faster.

Example

Consider two historical datasets with census data to link. The datasets are named `census_data_1980` and `census_data_1990`. The MultiIndex `candidate_pairs` contains the record pairs to compare. The record pairs are compared on the first name, last name, sex, date of birth, address, place, and income:

```
# initialise class
comp = recordlinkage.Compare()

# initialise similarity measurement algorithms
comp.string('first_name', 'name', method='jarowinkler')
comp.string('lastname', 'lastname', method='jarowinkler')
comp.exact('dateofbirth', 'dob')
comp.exact('sex', 'sex')
comp.string('address', 'address', method='levenshtein')
comp.exact('place', 'place')
comp.numeric('income', 'income')

# the method .compute() returns the DataFrame with the feature vectors.
comp.compute(candidate_pairs, census_data_1980, census_data_1990)
```

exact (*s1, s2, agree_value=1, disagree_value=0, missing_value=0, label=None*)
Compare the record pairs exactly.

This method initialises the exact similarity measurement between values. The similarity is 1 in case of agreement and 0 otherwise.

Parameters

- **s1** (*str or int*) – Field name to compare in left DataFrame.
- **s2** (*str or int*) – Field name to compare in right DataFrame.
- **agree_value** (*float, str, numpy.dtype*) – The value when two records are identical. Default 1. If 'values' is passed, then the value of the record pair is passed.
- **disagree_value** (*float, str, numpy.dtype*) – The value when two records are not identical.
- **missing_value** (*float, str, numpy.dtype*) – The value for a comparison with a missing value. Default 0.
- **label** (*label*) – The label of the column in the resulting dataframe.

string (*s1, s2, method='levenshtein', threshold=None, missing_value=0, label=None*)
Compute the (partial) similarity between strings values.

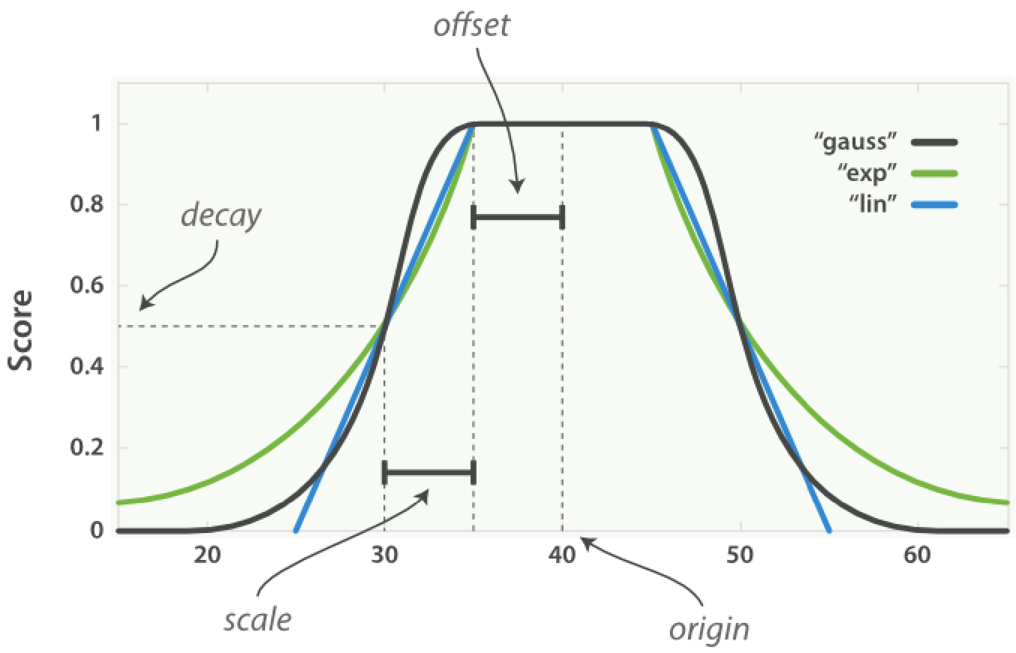
This method initialises the similarity measurement between string values. The implemented algorithms are: 'jaro','jarowinkler', 'levenshtein', 'damerau_levenshtein', 'qgram' or 'cosine'. In case of agreement, the similarity is 1 and in case of complete disagreement it is 0. The Python Record Linkage Toolkit uses the *jellyfish* package for the Jaro, Jaro-Winkler, Levenshtein and Damerau-Levenshtein algorithms.

Parameters

- **s1** (*str or int*) – The name or position of the column in the left DataFrame.
- **s2** (*str or int*) – The name or position of the column in the right DataFrame.
- **method** (*str, default 'levenshtein'*) – An approximate string comparison method. Options are [‘jaro’, ‘jarowinkler’, ‘levenshtein’, ‘damerau_levenshtein’, ‘qgram’, ‘cosine’, ‘smith_waterman’, ‘lcs’]. Default: ‘levenshtein’
- **threshold** (*float, tuple of floats*) – A threshold value. All approximate string comparisons higher or equal than this threshold are 1. Otherwise 0.
- **missing_value** (*numpy.dtype*) – The value for a comparison with a missing value. Default 0.
- **label** (*label*) – The label of the column in the resulting dataframe.

numeric (*s1, s2, method='linear', offset, scale, origin=0, missing_value=0, label=None*)
Compute the (partial) similarity between numeric values.

This method initialises the similarity measurement between numeric values. The implemented algorithms are: ‘step’, ‘linear’, ‘exp’, ‘gauss’ or ‘squared’. In case of agreement, the similarity is 1 and in case of complete disagreement it is 0. The implementation is similar with numeric comparing in ElasticSearch, a full-text search tool. The parameters are explained in the image below (source ElasticSearch, The Definitive Guide)



Parameters

- **s1** (*str or int*) – The name or position of the column in the left DataFrame.
- **s2** (*str or int*) – The name or position of the column in the right DataFrame.
- **method** (*float*) – The metric used. Options ‘step’, ‘linear’, ‘exp’, ‘gauss’ or ‘squared’. Default ‘linear’.
- **offset** (*float*) – The offset. See image above.

- **scale** (*float*) – The scale of the numeric comparison method. See the image above. This argument is not available for the 'step' algorithm.
- **origin** (*str*) – The shift of bias between the values. See image above.
- **missing_value** (*numpy.dtype*) – The value if one or both records have a missing value on the compared field. Default 0.
- **label** (*label*) – The label of the column in the resulting dataframe.

Note: Numeric comparing can be an efficient way to compare date/time variables. This can be done by comparing the timestamps.

geo (*lat1*, *lng1*, *lat2*, *lng2*, *method='linear'*, *offset*, *scale*, *origin=0*, *missing_value=0*, *label=None*)
Compute the (partial) similarity between WGS84 coordinate values.

Compare the geometric (haversine) distance between two WGS- coordinates. The similarity algorithms are 'step', 'linear', 'exp', 'gauss' or 'squared'. The similarity functions are the same as in `recordlinkage.comparing.Compare.numeric()`

Parameters

- **lat1** (*str or int*) – The name or position of the column in the left DataFrame.
- **lng1** (*str or int*) – The name or position of the column in the left DataFrame.
- **lat2** (*str or int*) – The name or position of the column in the right DataFrame.
- **lng2** (*str or int*) – The name or position of the column in the right DataFrame.
- **method** (*str*) – The metric used. Options 'step', 'linear', 'exp', 'gauss' or 'squared'. Default 'linear'.
- **offset** (*float*) – The offset. See Compare.numeric.
- **scale** (*float*) – The scale of the numeric comparison method. See Compare.numeric. This argument is not available for the 'step' algorithm.
- **origin** (*float*) – The shift of bias between the values. See Compare.numeric.
- **missing_value** (*numpy.dtype*) – The value for a comparison with a missing value. Default 0.
- **label** (*label*) – The label of the column in the resulting dataframe.

date (*self*, *s1*, *s2*, *swap_month_day=0.5*, *swap_months='default'*, *missing_value=0*, *label=None*)
Compute the (partial) similarity between date values.

Parameters

- **s1** (*str or int*) – The name or position of the column in the left DataFrame.
- **s2** (*str or int*) – The name or position of the column in the right DataFrame.
- **swap_month_day** (*float*) – The value if the month and day are swapped.
- **swap_months** (*list of tuples*) – A list of tuples with common errors caused by the translating of months into numbers, i.e. October is month 10. The format of the tuples is (month_good, month_bad, value). Default : swap_months = [(6, 7, 0.5), (7, 6, 0.5), (9, 10, 0.5), (10, 9, 0.5)]
- **missing_value** (*numpy.dtype*) – The value for a comparison with a missing value. Default 0.

- **label** (*label*) – The label of the column in the resulting dataframe.

compare_vectorized(*comp_func*, *labels_left*, *labels_right*, **args*, ***kwargs*)
Compute the similarity between values with a callable.

This method initialises the comparing of values with a custom function/callable. The function/callable should accept numpy.ndarray's.

Example

```
>>> comp = recordlinkage.Compare()
>>> comp.compare_vectorized(custom_callable, 'first_name', 'name')
>>> comp.compare(PAIRS, DATAFRAME1, DATAFRAME2)
```

Parameters

- **comp_func** (*function*) – A comparison function. This function can be a built-in function or a user defined comparison function. The function should accept numpy.ndarray's as first two arguments.
- **labels_left** (*label, pandas.Series, pandas.DataFrame*) – The labels, Series or DataFrame to compare.
- **labels_right** (*label, pandas.Series, pandas.DataFrame*) – The labels, Series or DataFrame to compare.
- ***args** – Additional arguments to pass to callable comp_func.
- ****kwargs** – Additional keyword arguments to pass to callable comp_func. (keyword 'label' is reserved.)
- **label** (*(list of) label(s)*) – The name of the feature and the name of the column. IMPORTANT: This argument is a keyword argument and can not be part of the arguments of comp_func.

compute(*pairs*, *x*, *x_link=None*)
Compare the records of each record pair.

Calling this method starts the comparing of records.

Parameters

- **pairs** (*pandas.MultiIndex*) – A pandas MultiIndex with the record pairs to compare. The indices in the MultiIndex are indices of the DataFrame(s) to link.
- **x** (*pandas.DataFrame*) – The DataFrame to link. If *x_link* is given, the comparing is a linking problem. If *x_link* is not given, the problem is one of deduplication.
- **x_link** (*pandas.DataFrame, optional*) – The second DataFrame.

Returns *pandas.DataFrame* – A pandas DataFrame with feature vectors, i.e. the result of comparing each record pair.

8.3.1 Migrating

Version 0.10 of the Python Record Linkage Toolkit uses a new API to compare record pairs. The new API uses a different syntax. Records are now compared after calling the *compute* method. Also, the *Compare* class is no longer initialized with the data and the record pairs. The data and record pairs are passed to the *compute* method. **The old procedure still works but will be removed in the future.**

Old (linking):

```
c = recordlinkage.Compare(candidate_links, df_a, df_b)

c.string('name_a', 'name_b', method='jarowinkler', threshold=0.85)
c.exact('sex', 'gender')
c.date('dob', 'date_of_birth')
c.string('str_name', 'streetname', method='damerau_levenshtein', threshold=0.7)
c.exact('place', 'placename')
c.numeric('income', 'income', method='gauss', offset=3, scale=3, missing_value=0.5)

# The comparison vectors
c.vectors
```

New (linking):

```
c = recordlinkage.Compare()

c.string('name_a', 'name_b', method='jarowinkler', threshold=0.85)
c.exact('sex', 'gender')
c.date('dob', 'date_of_birth')
c.string('str_name', 'streetname', method='damerau_levenshtein', threshold=0.7)
c.exact('place', 'placename')
c.numeric('income', 'income', method='gauss', offset=3, scale=3, missing_value=0.5)

# The comparison vectors
feature_vectors = c.compute(candidate_links, df_a, df_b)
```

Old (deduplication):

```
c = recordlinkage.Compare(candidate_links, df_a)

c.string('name_a', 'name_b', method='jarowinkler', threshold=0.85)
c.exact('sex', 'gender')
c.date('dob', 'date_of_birth')
c.string('str_name', 'streetname', method='damerau_levenshtein', threshold=0.7)
c.exact('place', 'placename')
c.numeric('income', 'income', method='gauss', offset=3, scale=3, missing_value=0.5)

# The comparison vectors
c.vectors
```

New (deduplication):

```
c = recordlinkage.Compare()

c.string('name_a', 'name_b', method='jarowinkler', threshold=0.85)
c.exact('sex', 'gender')
c.date('dob', 'date_of_birth')
c.string('str_name', 'streetname', method='damerau_levenshtein', threshold=0.7)
c.exact('place', 'placename')
c.numeric('income', 'income', method='gauss', offset=3, scale=3, missing_value=0.5)

# The comparison vectors
feature_vectors = c.compute(candidate_links, df_a)
```

8.4 Classification

Classification is the step in the record linkage process were record pairs are classified into matches, non-matches and possible matches [*Christen2012*]. Classification algorithms can be supervised or unsupervised (rougly speaking: with or without training data). Many of the algorithms need trainings data to classify the record pairs. Trainings data is data for which is known whether it is a match or not.

See also:

`exception LearningError`
Learning error

`class Classifier`
Base class for classification of records pairs.

This class contains methods for training the classifier. Distinguish different types of training, such as supervised and unsupervised learning.

`learn` (*comparison_vectors*, *match_index*, *return_type='index'*)
Train the classifier.

Parameters

- **`comparison_vectors`** (*`pandas.DataFrame`*) – The comparison vectors.
- **`match_index`** (*`pandas.MultiIndex`*) – The true matches.
- **`return_type`** (*`'index' (default), 'series', 'array'`*) – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

`predict` (*comparison_vectors*, *return_type='index'*)
Predict the class of the record pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

- **`comparison_vectors`** (*`pandas.DataFrame`*) – Dataframe with comparison vectors.
- **`return_type`** (*`'index' (default), 'series', 'array'`*) – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

`prob` (*comparison_vectors*, *return_type='series'*)
Compute the probabilities for each record pair.

For each pair of records, estimate the probability of being a match.

Parameters

- **`comparison_vectors`** (*`pandas.DataFrame`*) – The dataframe with comparison vectors.
- **`return_type`** (*`'series' or 'array'`*) – Return a pandas series or numpy array. Default 'series'.

Returns *pandas.Series or numpy.ndarray* – The probability of being a match for each record pair.

`class KMeansClassifier`(*match_cluster_center=None*, *nonmatch_cluster_center=None*, **args*, ***kwargs*)

KMeans classifier.

The K-means clusterings algorithm (wikipedia) partitions candidate record pairs into matches and non-matches. Each comparison vector belongs to the cluster with the nearest mean. The K-means algorithm does not need trainings data, but it needs two starting points (one for the matches and one for the non-matches). The K-means clustering problem is NP-hard.

Parameters

- **`match_cluster_center`** (*`list, numpy.array`*) – The center of the match cluster. The length of the list/array must equal the number of comparison variables.
- **`nonmatch_cluster_center`** (*`list, numpy.array`*) – The center of the non-match (distinct) cluster. The length of the list/array must equal the number of comparison variables.

`classifier`

sklearn.cluster.KMeans – The Kmeans cluster class in sklearn.

`match_cluster_center`

list, numpy.array – The center of the match cluster.

`nonmatch_cluster_center`

list, numpy.array – The center of the nonmatch (distinct) cluster.

Note: There are way better methods for linking records than the k-means clustering algorithm. However, this algorithm does not need trainings data and is useful to do an initial partition.

`learn`(*comparison_vectors*, *return_type='index'*)

Train the K-means classifier.

The K-means classifier is unsupervised and therefore does not need labels. The K-means classifier classifies the data into two sets of links and non- links. The starting point of the cluster centers are 0.05 for the non-matches and 0.95 for the matches.

Parameters

- **`comparison_vectors`** (*`pandas.DataFrame`*) – The dataframe with comparison vectors.
- **`return_type`** (*`'index' (default), 'series', 'array'`*) – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns *pandas.MultiIndex, pandas.Series or numpy.ndarray* – The prediction (see also the argument 'return_type')

`predict` (*comparison_vectors*, *return_type='index'*)
Predict the class of the record pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

- **`comparison_vectors`** (*`pandas.DataFrame`*) – Dataframe with comparison vectors.
- **`return_type`** (*`'index' (default), 'series', 'array'`*) – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

`class LogisticRegressionClassifier` (*coefficients=None*, *intercept=None*)
Logistic Regression Classifier.

This classifier is an application of the logistic regression model (wikipedia). The classifier partitions candidate record pairs into matches and non-matches.

Parameters

- **`coefficients`** (*`list, numpy.array`*) – The coefficients of the logistic regression.
- **`intercept`** (*`float`*) – The interception value.

`classifier`
sklearn.linear_model.LogisticRegression – The Logistic regression classifier in sklearn.

`coefficients`
list – The coefficients of the logistic regression.

`intercept`
float – The interception value.

`learn` (*comparison_vectors*, *match_index*, *return_type='index'*)
Train the classifier.

Parameters

- **`comparison_vectors`** (*`pandas.DataFrame`*) – The comparison vectors.
- **`match_index`** (*`pandas.MultiIndex`*) – The true matches.
- **`return_type`** (*`'index' (default), 'series', 'array'`*) – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

`predict` (*comparison_vectors*, *return_type='index'*)
Predict the class of the record pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

- **`comparison_vectors`** (*`pandas.DataFrame`*) – Dataframe with comparison vectors.
- **`return_type`** (*`'index' (default), 'series', 'array'`*) – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

`prob`(*comparison_vectors*, *return_type='series'*)

Compute the probabilities for each record pair.

For each pair of records, estimate the probability of being a match.

Parameters

- **`comparison_vectors`** (*`pandas.DataFrame`*) – The dataframe with comparison vectors.
- **`return_type`** (*`'series' or 'array'`*) – Return a pandas series or numpy array. Default 'series'.

Returns *pandas.Series or numpy.ndarray* – The probability of being a match for each record pair.

class `NaiveBayesClassifier`(*alpha=1.0*)

Naive Bayes Classifier.

The Naive Bayes classifier (wikipedia) partitions candidate record pairs into matches and non-matches. The classifier is based on probabilistic principles. The Naive Bayes classification method is proven to be mathematical equivalent with the Fellegi and Sunter model.

Parameters **`log_prior`** (*`list, numpy.array`*) – The log propabaility of each class.

`classifier`

sklearn.linear_model.LogisticRegression – The Logistic regression classifier in sklearn.

`coefficients`

list – The coefficients of the logistic regression.

`intercept`

float – The interception value.

Parameters **`alpha`** (*`float`*) – Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

`learn`(*comparison_vectors*, *match_index*, *return_type='index'*)

Train the classifier.

Parameters

- **`comparison_vectors`** (*`pandas.DataFrame`*) – The comparison vectors.
- **`match_index`** (*`pandas.MultiIndex`*) – The true matches.
- **`return_type`** (*`'index' (default), 'series', 'array'`*) – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series

with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

`predict` (*comparison_vectors*, *return_type='index'*)

Predict the class of the record pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

- **`comparison_vectors`** (`pandas.DataFrame`) – Dataframe with comparison vectors.
- **`return_type`** (`'index' (default), 'series', 'array'`) – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

`prob` (*comparison_vectors*, *return_type='series'*)

Compute the probabilities for each record pair.

For each pair of records, estimate the probability of being a match.

Parameters

- **`comparison_vectors`** (`pandas.DataFrame`) – The dataframe with comparison vectors.
- **`return_type`** (`'series' or 'array'`) – Return a pandas series or numpy array. Default 'series'.

Returns *pandas.Series or numpy.ndarray* – The probability of being a match for each record pair.

`class SVMClassifier`

Support Vector Machines

The Support Vector Machine classifier (wikipedia) partitions candidate record pairs into matches and non-matches. This implementation is a non-probabilistic binary linear classifier. Support vector machines are supervised learning models. Therefore, the SVM classifiers needs training-data.

`learn` (*comparison_vectors*, *match_index*, *return_type='index'*)

Train the classifier.

Parameters

- **`comparison_vectors`** (`pandas.DataFrame`) – The comparison vectors.
- **`match_index`** (`pandas.MultiIndex`) – The true matches.
- **`return_type`** (`'index' (default), 'series', 'array'`) – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

predict(*comparison_vectors*, *return_type='index'*)
Predict the class of the record pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – Dataframe with comparison vectors.
- **return_type** (*'index' (default), 'series', 'array'*) – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

class FellegiSunter(*random_decision_rule=False*)
Fellegi and Sunter framework.

Base class for probabilistic classification of records pairs with the Fellegi and Sunter (1969) framework.

learn(*comparison_vectors*, *match_index*, *return_type='index'*)
Train the classifier.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The comparison vectors.
- **match_index** (*pandas.MultiIndex*) – The true matches.
- **return_type** (*'index' (default), 'series', 'array'*) – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

predict(*comparison_vectors*, *return_type='index'*)
Predict the class of the record pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – Dataframe with comparison vectors.
- **return_type** (*'index' (default), 'series', 'array'*) – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

`prob` (*comparison_vectors*, *return_type='series'*)
Compute the probabilities for each record pair.

For each pair of records, estimate the probability of being a match.

Parameters

- **`comparison_vectors`** (*`pandas.DataFrame`*) – The dataframe with comparison vectors.
- **`return_type`** (*`'series' or 'array'`*) – Return a pandas series or numpy array. Default 'series'.

Returns *pandas.Series or numpy.ndarray* – The probability of being a match for each record pair.

`class ECMClassifier` (**args*, ***kwargs*)
Expectation/Conditional Maxisation vlassifier.

[EXPERIMENTAL] Expectation/Conditional Maximisation algorithm used as classifier. This probabilistic record linkage algorithm is used in combination with Fellegi and Sunter model.

`learn` (*comparison_vectors*, *init='jaro'*, *return_type='index'*)
Train the algorithm.

Train the Expectation-Maximisation classifier. This method is well- known as the ECM-algorithm implementation in the context of record linkage.

Parameters

- **`comparison_vectors`** (*`pandas.DataFrame`*) – The dataframe with comparison vectors.
- **`params_init`** (*`dict`*) – A dictionary with initial parameters of the ECM algorithm (optional).
- **`return_type`** (*`'index' (default), 'series', 'array'`*) – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

`predict` (*comparison_vectors*, *return_type='index'*, **args*, ***kwargs*)
Predict the class of reord pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

- **`comparison_vectors`** (*`pandas.DataFrame`*) – The dataframe with comparison vectors.
- **`return_type`** (*`'index' (default), 'series', 'array'`*) – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

Note: Prediction is risky for this unsupervised learning method. Be aware that the sample from the population is valid.

prob(*comparison_vectors*)
Compute the probabilities for each record pair.

For each pair of records, estimate the probability of being a match.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The dataframe with comparison vectors.
- **return_type** (*'series' or 'array'*) – Return a pandas series or numpy array. Default 'series'.

Returns *pandas.Series or numpy.ndarray* – The probability of being a match for each record pair.

8.5 Evaluation

Evaluation of classifications plays an important role in record linkage. Express your classification quality in terms accuracy, recall and F-score based on `true positives`, `false positives`, `true negatives` and `false negatives`.

reduction_ratio(*links_pred*, **total*)
Compute the reduction ratio.

The reduction ratio is 1 minus the ratio candidate matches and the maximum number of pairs possible.

Parameters

- **links_pred** (*int, pandas.MultiIndex*) – The number of candidate record pairs or the pandas.MultiIndex with record pairs.
- ***total** (*pandas.DataFrame object(s)*) – The DataFrames are used to compute the full index size with the full_index_size function.

Returns *float* – The reduction ratio.

max_pairs(*shape*)
[DEPRECATED] Compute the maximum number of record pairs possible.

full_index_size(**args*)
Compute the number of records in a full index.

Compute the number of records in a full index without building the index itself. The result is the maximum number of record pairs possible. This function is especially useful in measures like the *reduction_ratio*.

Deduplication: Given a DataFrame A with length N, the full index size is N*(N-1)/2. Linking: Given a DataFrame A with length N and a DataFrame B with length M, the full index size is N*M.

Parameters ***args** (*int, pandas.MultiIndex, pandas.Series, pandas.DataFrame*) – A pandas object or a int representing the length of a dataset to link. When there is one argument, it is assumed that the record linkage is a deduplication process.

Examples

Use integers: >>> full_index_size(10) # deduplication: 45 pairs >>> full_index_size(10, 10) # linking: 100 pairs

or pandas objects >>> full_index_size(DF) # deduplication: len(DF)*(len(DF)-1)/2 pairs >>> full_index_size(DF, DF) # linking: len(DF)*len(DF) pairs

`true_positives`(*links_true*, *links_pred*)

Count the number of True Positives.

Returns the number of correctly predicted links, also called the number of True Positives (TP).

Parameters

- **`links_true`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The true (or actual) links.
- **`links_pred`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The predicted links.

Returns *int* – The number of correctly predicted links.

`true_negatives`(*links_true*, *links_pred*, *total*)

Count the number of True Negatives.

Returns the number of correctly predicted non-links, also called the number of True Negatives (TN).

Parameters

- **`links_true`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The true (or actual) links.
- **`links_pred`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The predicted links.
- **`total`** (*`int, pandas.MultiIndex`*) – The count of all record pairs (both links and non-links). When the argument is a pandas.MultiIndex, the length of the index is used.

Returns *int* – The number of correctly predicted non-links.

`false_positives`(*links_true*, *links_pred*)

Count the number of False Positives.

Returns the number of incorrect predictions of true non-links. (true non- links, but predicted as links). This value is known as the number of False Positives (FP).

Parameters

- **`links_true`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The true (or actual) links.
- **`links_pred`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The predicted links.

Returns *int* – The number of false positives.

`false_negatives`(*links_true*, *links_pred*)

Count the number of False Negatives.

Returns the number of incorrect predictions of true links. (true links, but predicted as non-links). This value is known as the number of False Negatives (FN).

Parameters

- **`links_true`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The true (or actual) links.
- **`links_pred`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The predicted links.

Returns *int* – The number of false negatives.

`confusion_matrix`(*links_true*, *links_pred*, *total=None*)

Compute the confusion matrix.

The confusion matrix is of the following form:

	Predicted Positives	Predicted Negatives
True Positives	True Positives (TP)	False Negatives (FN)
True Negatives	False Positives (FP)	True Negatives (TN)

The confusion matrix is an informative way to analyse a prediction. The matrix can used to compute measures like precision and recall. The count of true prositives is [0,0], false negatives is [0,1], true negatives is [1,1] and false positives is [1,0].

Parameters

- **`links_true`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The true (or actual) links.
- **`links_pred`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The predicted links.
- **`total`** (*`int, pandas.MultiIndex`*) – The count of all record pairs (both links and non-links). When the argument is a pandas.MultiIndex, the length of the index is used. If the total is None, the number of True Negatives is not computed. Default None.

Returns *numpy.array* – The confusion matrix with TP, TN, FN, FP values.

Note: The number of True Negatives is computed based on the total argument. This argument is the number of record pairs of the entire matrix.

`precision`(*links_true*, *links_pred*)

Compute the precision.

The precision is given by TP/(TP+FP).

Parameters

- **`links_true`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The true (or actual) collection of links.
- **`links_pred`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The predicted collection of links.

Returns *float* – The precision

`recall`(*links_true*, *links_pred*)

Compute the recall/sensitivity.

The recall is given by TP/(TP+FN).

Parameters

- **`links_true`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The true (or actual) collection of links.
- **`links_pred`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The predicted collection of links.

Returns *float* – The recall

`accuracy` (*links_true*, *links_pred*, *total*)

Compute the accuracy.

The accuracy is given by (TP+TN)/(TP+FP+TN+FN).

Parameters

- **`links_true`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The true (or actual) collection of links.
- **`links_pred`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The predicted collection of links.
- **`total`** (*`int, pandas.MultiIndex`*) – The count of all record pairs (both links and non-links). When the argument is a pandas.MultiIndex, the length of the index is used.

Returns *float* – The accuracy

`specificity` (*links_true*, *links_pred*, *total*)

Compute the specificity.

The specificity is given by TN/(FP+TN).

Parameters

- **`links_true`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The true (or actual) collection of links.
- **`links_pred`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The predicted collection of links.
- **`total`** (*`int, pandas.MultiIndex`*) – The count of all record pairs (both links and non-links). When the argument is a pandas.MultiIndex, the length of the index is used.

Returns *float* – The specificity

`fscore` (*links_true*, *links_pred*)

Compute the F-score.

The F-score is given by 2*(precision*recall)/(precision+recall).

Parameters

- **`links_true`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The true (or actual) collection of links.
- **`links_pred`** (*`pandas.MultiIndex, pandas.DataFrame, pandas.Series`*) – The predicted collection of links.

Returns *float* – The fscore

Note: If there are no pairs predicted as links, this measure will raise a ZeroDivisionError.

8.6 Datasets

The Python Record Linkage Toolkit contains several open public datasets. Four datasets were generated by the developers of Febrl. In the future, we are developing tools to generate your own datasets.

`load_krebsregister`(*block=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], missing_values=None, shuffle=True*)

Dataset 'Krebsregister'

This dataset of comparison patterns was obtained in a epidemiological cancer study in Germany. The comparison patterns were created by the Institute for Medical Biostatistics, Epidemiology and Informatics (IMBEI) and the University Medical Center of Johannes Gutenberg University (Mainz, Germany). The dataset is available for research online.

> "The records represent individual data including first and family name, sex, date of birth and postal code, which were collected through iterative insertions in the course of several years. The comparison patterns in this data set are based on a sample of 100.000 records dating from 2005 to 2008. Data pairs were classified as "match" or "non-match" during an extensive manual review where several documentarists were involved. The resulting classification formed the basis for assessing the quality of the registry's own record linkage procedure.
>
> In order to limit the amount of patterns a blocking procedure was applied, which selects only record pairs that meet specific agreement conditions. The results of the following six blocking iterations were merged together:
>
> 1. Phonetic equality of first name and family name, equality of date of birth.
> 2. Phonetic equality of first name, equality of day of birth.
> 3. Phonetic equality of first name, equality of month of birth.
> 4. Phonetic equality of first name, equality of year of birth.
> 5. Equality of complete date of birth.
> 6. Phonetic equality of family name, equality of sex.
>
> This procedure resulted in 5.749.132 record pairs, of which 20.931 are matches. The data set is split into 10 blocks of (approximately) equal size and ratio of matches to non-matches."

Parameters

- **`block`** (*`int, list`*) – An integer or a list with integers between 1 and 10. The blocks are the blocks explained in the description.
- **`missing_values`** (*`object, int, float`*) – The value of the missing values. Default NaN.
- **`shuffle`** (*`bool`*) – Shuffle the record pairs. Default True.

Returns *(pandas.DataFrame, pandas.MultiIndex)* – A data frame with comparison vectors and a multi index with the indices of the matches.

`load_febrl1`(*return_links=False*)

FEBRL dataset 1

The Freely Extensible Biomedical Record Linkage (Febrl) package was distributed with a dataset generator and four datasets generated with the generator. This functions returns the first Febrl dataset as a pandas DataFrame.

> *"This data set contains 1000 records (500 original and 500 duplicates, with exactly one duplicate per original record."*

Parameters **`return_links`** (*`bool`*) – When True, the function returns also the true links.

Returns *pandas.DataFrame* – A pandas DataFrame with Febrl dataset1.csv. When return_links is True, the function returns also the true links.

load_febrl2(*return_links=False*)
FEBRL dataset 2

The Freely Extensible Biomedical Record Linkage (Febrl) package was distributed with a dataset generator and four datasets generated with the generator. This functions returns the second Febrl dataset as a pandas DataFrame.

"This data set contains 5000 records (4000 originals and 1000 duplicates), with a maximum of 5 duplicates based on one original record (and a poisson distribution of duplicate records). Distribution of duplicates: 19 originals records have 5 duplicate records 47 originals records have 4 duplicate records 107 originals records have 3 duplicate records 141 originals records have 2 duplicate records 114 originals records have 1 duplicate record 572 originals records have no duplicate record"

Parameters **return_links** (*bool*) – When True, the function returns also the true links.

Returns *pandas.DataFrame* – A pandas DataFrame with Febrl dataset2.csv. When return_links is True, the function returns also the true links.

load_febrl3(*return_links=False*)
FEBRL dataset 3

The Freely Extensible Biomedical Record Linkage (Febrl) package was distributed with a dataset generator and four datasets generated with the generator. This functions returns the third Febrl dataset as a pandas DataFrame.

"This data set contains 5000 records (2000 originals and 3000 duplicates), with a maximum of 5 duplicates based on one original record (and a Zipf distribution of duplicate records). Distribution of duplicates: 168 originals records have 5 duplicate records 161 originals records have 4 duplicate records 212 originals records have 3 duplicate records 256 originals records have 2 duplicate records 368 originals records have 1 duplicate record 1835 originals records have no duplicate record"

Parameters **return_links** (*bool*) – When True, the function returns also the true links.

Returns *pandas.DataFrame* – A pandas DataFrame with Febrl dataset3.csv. When return_links is True, the function returns also the true links.

load_febrl4(*return_links=False*)
FEBRL dataset 4

The Freely Extensible Biomedical Record Linkage (Febrl) package was distributed with a dataset generator and four datasets generated with the generator. This functions returns the fourth Febrl dataset as a pandas DataFrame.

"Generated as one data set with 10000 records (5000 originals and 5000 duplicates, with one duplicate per original), the originals have been split from the duplicates, into dataset4a.csv (containing the 5000 original records) and dataset4b.csv (containing the 5000 duplicate records) These two data sets can be used for testing linkage procedures."

Parameters **return_links** (*bool*) – When True, the function returns also the true links.

Returns *(pandas.DataFrame, pandas.DataFrame)* – A pandas DataFrame with Febrl dataset4a.csv and a pandas dataframe with Febrl dataset4b.csv. When return_links is True, the function returns also the true links.

binary_vectors(*n, n_match, m=[0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9], u=[0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1], random_state=None*)
Generate random binary comparison vectors

This function is used to generate random comparison vectors. The result of each comparison is a binary value (0 or 1).

Parameters

- **n** (*int*) – The total number of comparison vectors.
- **n_match** (*int*) – The number of matching record pairs.
- **m** (*list, default [0.9] * 8, optional*) – A list of m probabilities of each partially identifying variable. The m probability is the probability that an identifier in matching record pairs agrees.
- **u** (*list, default [0.9] * 8, optional*) – A list of u probabilities of each partially identifying variable. The u probability is the probability that an identifier in non-matching record pairs agrees.
- **random_state** (*int or numpy.random.RandomState, optional*) – Seed for the random number generator with an integer or numpy RandomState object.

Returns *pandas.DataFrame* – A dataframe with comparison vectors.

CHAPTER 9

Contributing

Thanks for your interest in contributing to the Python Record Linkage Toolkit. There is a lot of work to do. See Github for the contributors to this package.

The workflow for contributing is as follows:

- clone https://github.com/J535D165/recordlinkage.git
- Make a branch with your modifications/contributions
- Write tests
- Run all tests
- Do a pull request

9.1 Testing

Install Nose:

```
pip install nose parameterized
```

Run the following command to test the package

```
nosetests
```

9.2 Performance

Performance is very important in record linkage. The performance is monitored for all serious modifications of the core API. The performance monitoring is performed with Airspeed Velocity (asv).

Install Airspeed Velocity:

```
pip install asv
```

Run the following command from the root of the repository to test the performance of the current version of the package:

```
asv run
```

Run the following command to test all versions since tag v0.6.0

```
asv run --skip-existing-commits v0.6.0..master
```

CHAPTER 10

Release notes

10.1 Version 0.11.0

- The submodule ‘standardise’ is renamed. The new name is ‘preprocessing’. The submodule ‘standardise’ will get deprecated in a next version.
- Deprecation errors were not visible for many users. In this version, the errors are better visible.
- Improved and new logs for indexing, comparing and classification.
- Faster comparing of string variables. Thanks Joel Becker.
- Changes make it possible to pickle Compare and Index objects. This makes it easier to run code in parallel. Tests were added to ensure that pickling remains possible.
- Important change. MultiIndex objects with many record pairs were split into pieces to lower memory usage. In this version, this automatic splitting is removed. Please split the data yourself.
- Integer indexing. Blog post will follow on this.
- The metrics submodule has changed heavily. This will break with the previous version.
- repr() and str() will return informative information for index and compare objects.
- It is possible to use abbreviations for string similarity methods. For example ‘jw’ for the Jaro-Winkler method.
- The FEBRL dataset loaders can now return the true links as a pandas.MultIndex for each FEBRL dataset. This option is disabled by default. See the [FEBRL datasets][febrl_datasets] for details.
- Fix issue with automatic recognision of license on Github.
- Various small improvements.

[febrl_datasets]: http://recordlinkage.readthedocs.io/en/latest/ref-datasets.html#recordlinkage.datasets.load_febrl1

Note: In the next release, the Pairs class will get removed. Migrate now.

10.2 Version 0.10.1

- print statement in the geo compare algorithm removed.
- String, numeric and geo compare functions now raise directly when an incorrect algorithm name is passed.
- Fix unit test that failed on Python 2.7.

10.3 Version 0.10.0

- A new compare API. The new Compare class no longer takes the datasets and pairs as arguments. The actual computation is now performed when calling *.compute(PAIRS, DF1, DF2)*. The documentation is updated as well, but still needs improvement.
- Two new string similarity measures are added: Smith Waterman (smith_waterman) and Longest Common Substring (lcs). Thanks to Joel Becker and Jillian Anderson from the Networks Lab of the University of Waterloo.
- Added and/or updated a large amount of unit tests.
- Various small improvements.

10.4 Version 0.9.0

- A new index API. The new index API is no longer a single class (`recordlinkage.Pairs(...)`) with all the functionality in it. The new API is based on Tensorflow and FEBRL. With the new structure, it easier to parallise the record linkage process. In future releases, this will be implemented natively. See the reference page for more information and migrating.
- Significant speed improvement of the Sorted Neighbourhood Indexing algorithm. Thanks to @perryvais (PR #32).
- The function `binary_comparisons` is renamed. The new name of the function is `binary_vectors`. Documentation added to RTD.
- Added unit tests to test the generation of random comparison vectors.
- Logging module added to separate module logs from user logs. The implementation is based on Tensorflow.

10.5 Version 0.8.1

- Issues solved with rendering docs on ReadTheDocs. Still not clear what is going on with the `autodoc_mock_imports` in the sphinx conf.py file. Maybe a bug in sphinx.
- Move six to dependencies.
- The reference part of the docs is split into separate subsections. This makes the reference better readable.
- The landing page of the docs is slightly changed.

10.6 Version 0.8.0

- Add additional arguments to the function that downloads and loads the krebsregister data. The argument `missing_values` is used to fill missing values. Default: nothing is done. The argument `shuffle` is used to shuffle the records. Default is True.
- Remove the lastest traces of the old package name. The new package name is 'Python Record Linkage Toolkit'
- Better error messages when there are only matches or non-matches are passed to train the classifier.
- Add AirSpeedVelocity tests to test the performance.
- Compare for deduplication fixed. It was broken.
- Parameterized tests for the `Compare` class and its algorithms. Making use of `nose-parameterized` module.
- Update documentation about contributing.
- Bugfix/improvement when blocking on multiple columns with missing values.
- Fix bug #29 (https://github.com/J535D165/recordlinkage/issues/29). Package not working with pandas 0.18 and 0.17. Dropped support pandas 0.17 and fixed support for 0.18. Also added multi-dendency tests for TravisCI.
- Support for dedicated deduplication algorithms
- Special algorithm for full index in case of finding duplicates. Performce is 100x better.
- Function `max_number_of_pairs` to get the maximum number of pairs.
- `low_memory` for compare class.
- Improved performance in case of comparing a large number of record pairs.
- New documentation about custom algorithms
- New documentation about the use of classifiers.
- Possible to compare arrays and series directly without using labels.
- Make a dataframe with random comparison vectors with the `binary_comparisons` in the `recordlinkage.datasets.random` module.
- Set KMeans cluster centers by hand.
- Various documentation updates and improvements.
- Jellyfish is now a required dependency. Fixes bug #30 (https://github.com/J535D165/recordlinkage/issues/30).
- Added `tox.ini` to test packaging and installation of package.
- Drop requirements.txt file.
- Many small fixes and changes. Most of the changes cover the `Compare` module. Especially label handling is improved.

10.7 Version 0.7.2

- Incorrect name of the Levenshtein method in the string comparison method fixed.

10.8 Version 0.7.1

- Fix the generation of docs on ReadTheDocs.
- Installation issue fixed. Packages not found.
- Import issues solved.

10.9 Version 0.7.0

- Rename the package into 'Python Record Linkage Toolkit'
- Remove `similar_values` function
- Remove gender imputation tool
- Updated algorithms for comparing numberic variables. The new algorithms can compute the similarity with kernels like gaussian, linear, squared and exponential. Tests for these numeric comparison algorithms are included.
- Better NaN handling for compare functions.
- Algorithm added to compare dates.
- Add tests for date comparing.
- Divide the `Compare` class into two classes.
- Add documentation about performance tricks and concepts.
- Replace the comparison algorithms to a submodule.
- Include six in the package
- Drop `requests` module and use builtin Python functions.
- Add metaphone phonetic algorithm.
- Add match rating string comparing algorithm.
- Manual parameter handling for logistic regression. The attributes are `coefficients` and `intercept`.
- Drop class `BernoulliNBClassifier`.
- Various documentation updates.
- Many small other updates.

10.10 Version 0.6.0

- Reformatting the code such that it follows PEP8.
- Add Travis-CI and codecov support.
- Switch to distributing wheels.
- Fix bugs with depreciated pandas functions. `__sub__` is no longer used for computing the difference of Index objects. It is now replaced by `INDEX.difference(OTHER_INDEX)`.
- Exclude pairs with NaN's on the index-key in Q-gram indexing.
- Add tests for krebsregister dataset.

- Fix Python3 bug on krebsregister dataset.
- Improve unicode handling in phonetic encoding functions.
- Strip accents with the `clean` function.
- Add documentation
- Bug for random indexing with incorrect arguments fixed and tests added.
- Improved deployment workflow
- And much more

10.11 Version 0.5.0

- Batch comparing added. Signifant speed improvement.
- rldatasets are now included in the package itself.
- Added an experimental gender imputation tool.
- Blocking and SNI skip missing values
- No longer need for different index names
- FEBRL datasets included
- Unit tests for indexing and comparing improved
- Documentation updated

10.12 Version 0.4.0

- Fixes a serious bug with deduplication.
- Fixes undesired behaviour for sorted neighbourhood indexing with missing values.
- Add new datasets to the package like Febrl datasets
- Move Krebsregister dataset to this package.
- Improve and add some tests
- Various documentation updates

10.13 Version 0.3.0

- Total restructure of compare functions (The end of changing the API is close to now.)
- Compare method `numerical` is now named `numeric` and `fuzzy` is now named `string`.
- Add haversine formula to compare geographical records.
- Use numexpr for computing numeric comparisons.
- Add step, linear and squared comparing.
- Add eye index method.
- Improve, update and add new tests.

- Remove iterative indexing functions.
- New add chunks for indexing functions. These chunks are defined in the class Pairs. If chunks are defined, then the indexing functions returns a generator with an Index for each element.
- Update documentation.
- Various bug fixes.

10.14 Version 0.2.0

- Full Python3 support
- Update the parameters of the Logistic Regression Classifier manually. In literature, this is often denoted as the ‘deterministic record linkage’.
- Expectation/Conditional Maximization algorithm completely rewritten. The performance of the algorithm is much better now. The algorithm is still experimental.
- New string comparison metrics: Q-gram string comparing and Cosine string comparing.
- New indexing algorithm: Q-gram indexing.
- Several internal tests.
- Updated documentation.
- BernoulliNBClassifier is now named NaiveBayesClassifier. No changes to the algorithm.
- Arguments order in compare functions corrected.
- Function to clean phone numbers
- Return the result of the classifier as index, numpy array or pandas series.
- Many bug fixes

10.15 Version 0.1.0

- Official release

Bibliography

[christen2012] Christen, P. (2012). Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection. Springer Science & Business Media.

[christen2008] Christen, P. (2008). Febrl - A Freely Available Record Linkage System with a Graphical User Interface.

[Christen2012] Christen, Peter. 2012. Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection. Springer Science & Business Media.

Index